

# A Domain Specific Language for Dynamic Interest Management within Virtual Environments

Thesis by  
Sam Aaron

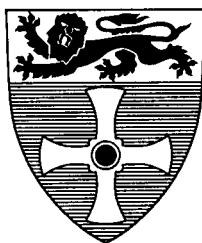
In Partial Fulfillment of the Requirements  
for the Degree of  
Doctor of Philosophy

NEWCASTLE UNIVERSITY LIBRARY

206 53475 1

Thesis L8723

UNIVERSITY OF  
NEWCASTLE UPON TYNE



University of Newcastle upon Tyne  
Newcastle upon Tyne, UK

2008

(Submitted August 2007)

© 2008

Sam Aaron

All Rights Reserved

I met myself and me,  
where I had been waiting so patiently.

We sat by the pool,  
where the water was cool,  
and jumped in,  
for a swim,  
*"I'm Free!"*

**To the Ruby Community.**



# Abstract

Interest management is a widely used term within the area of virtual environments. It is so widely used that there even exist many synonyms for the concept. Thus both the terminology, and meaning of the concept are currently not well defined. The typical aim of interest management techniques within virtual environments has been to increase scalability. However, this thesis argues that the concept of interest management should not be so tightly coupled with the goal of scalable virtual environments, but be a concept in its own right, i.e. *the management of interests*.

The main focus of this thesis is the representation of expressions of interest. The various techniques for expressing interest are surveyed and evaluated, providing the basis for the research into a suitable representation. This representation is achieved in two stages.

The first part of this thesis introduces a novel dynamic interest management technique based upon set theory. It describes how it is expressive enough to implement most of the static interest management techniques currently available such as categorisation, locales, and interacting locales. By de-coupling the logic that implements these interests from the virtual environment, it can also describe how interests can be changed during the virtual environment's execution, thus making the technique dynamic. Enforcing and denying interests is also considered, allowing for the enforcement of interests integral to the requirements of the virtual environment. An example of this is denying the user the ability to be interested in artefacts that aren't visible. The new approach presented is implemented with SQL, and evaluated.

The second part of this thesis focusses on the limitations of using SQL as an implementation language, focussing on issues of readability and succinctness and a lack of any abstraction mechanisms. Overcoming these limitations is treated as the primary design goal for a new domain specific language for representing interests. The thesis introduces this language, Wish, and evaluates it within the domain, demonstrating that it is as expressive as SQL yet is more readable, conceptually succinct and allows for arbitrary abstraction of complexity.

# Acknowledgements

An undertaking such as a Ph.D. is something that's probably only really possible with massive support from friends and family, and this was definitely the case for me. Without the wonderful support I have received over these years there is little doubt that I would have struggled to muster the strength to complete this research.

Firstly, I'd like to express my deepest respect and gratitude to Susanna who hasn't left my side throughout all of this - despite the way I dealt (or on occasion failed to deal) with the many challenges of the Ph.D. process. Susanna is truly amazing, and I'm so happy and proud to share my life with her. It's a struggle to find suitable words to express my appreciation for her support. This deep, powerful, wordless feeling of gratitude is also the case for my parents: Mum and David. Together, they have taught me such a lot about living, happiness and the importance of the small things in life. It's also a pleasure for me to thank my fellow inmate Chris Fowler for his companionship, friendship, and endless time for me. When I needed someone he was always there, and again it's hard to explain how important that was. *"We did it!"*

The people I have mentioned are really only the first few of a long list of important and special people in my life that have helped me in their own way with this undertaking. There is my sister Charlotte who exhibits a care and passion for life I can only look up to. There is my wonderful Nan who was always so proud of me, and of course the memories of my Granddad which have, and continue to be by my side. Thanks also to my supervisor Professor Paul Watson for always having an open door for me. I also need to thank my good friends Peter & Stephi, Paul Robinson, John Broderick, Will Stephenson, Kim Beerden, Massimo Strano, Adam Barker, Martin Ellis, Jake Wu, Jenna & Ben, John Colquhoun, Amy and Mitch, Simon Gamester, Hendrik Volkmer, Lee Irving, Dave Cooper, Iain Wood, Simon Woodman, Alex Cavanagh, Nat & Laura and many others. Thank you, all of you.

# Nomenclature

$N$	The total number of users in a virtual environment. ....	8
$e$	The average number of events a user creates per unit of time. ....	8
$\mu$	The average size of a message generated by a virtual environment. ....	8
$E$	The total number of events generated per unit of time within a virtual environment. ....	8
$M$	The total number of messages generated per unit of time by a virtual environment server. ....	8
$b$	The bandwidth needed per unit of time by a virtual environment to transmit messages. ....	8
$\sigma$	Ratio of artefacts that are interesting. ....	10
$\mathcal{U}$	The set of all artefacts within a virtual environment. ....	50
$\mathcal{I}$	The set of all interesting artefacts within a virtual environment. ....	50
$x$	A given artefact. ....	51
$A$	A relative artefact. ....	60
$A'$	A relative virtual artefact. ....	55
$\mathbb{I}(x)$	The condition which denotes whether or not $x$ is an interesting artefact. ....	51
$\mathbb{D}(x)$	The condition which denotes whether or not $x$ is a member of a given derived set. ....	53
$O$	The set of all essential artefacts. ....	62

# Abbreviations

<b>CRG</b>	Communications Research Group (Nottingham University)
<b>CVE</b>	Collaborative Virtual Environment, or CRG Virtual Environment
<b>DARPA</b>	Defence Advanced Research Projects Agency
<b>DIS</b>	Distributed Interactive Simulation
<b>DSL</b>	Domain Specific Language
<b>DIVE</b>	Distributed Interactive virtual environment
<b>HLA-DDM</b>	High Level Architecture Data Distribution Management
<b>IM</b>	Interest Management
<b>MASSIVE</b>	Model, Architecture and System for Spatial Interaction In Virtual Environments
<b>MMOG</b>	Massive Multiplayer Online Game
<b>MUD</b>	Multi User Dungeon
<b>NPSNET</b>	Navel Postgraduate School Network.
<b>ORM</b>	Object Relational Mapper
<b>SIMNET</b>	Simulation Network
<b>VE</b>	Virtual Environment
<b>VITA</b>	Visual Interaction Tool for Archaeology

# Contents

<b>Abstract</b>	<b>v</b>
<b>1 Introduction</b>	<b>5</b>
1.1 Limitations in Virtual Environments . . . . .	5
1.1.1 Scalability . . . . .	6
1.1.2 Adaptability . . . . .	7
1.2 A Mathematical Model of the Relationship between Number of Messages and Users within a Virtual Environment . . . . .	7
1.3 Approaches to Scalability . . . . .	9
1.3.1 Improving System Resources and Efficiency . . . . .	9
1.3.2 Increasing Application Efficiency . . . . .	10
1.4 Approaches to Adaptability . . . . .	11
1.5 Dynamic Interest Management for Adaptable Virtual Environments . . . . .	11
1.6 A Scenario . . . . .	12
1.6.1 Scenario analysis . . . . .	14
1.7 Contributions . . . . .	15
1.8 Thesis structure . . . . .	15
<b>2 Literature Survey</b>	<b>16</b>
2.1 Virtual Environments . . . . .	16
2.1.1 Definition . . . . .	16
2.1.1.1 Range of Definitions . . . . .	17
2.1.1.2 Use of Modifiers . . . . .	18
2.1.1.3 New Definition . . . . .	19
2.1.2 Origins . . . . .	21
2.1.3 Range of Usage . . . . .	22
2.1.3.1 MUDS . . . . .	22
2.1.3.2 War Simulations . . . . .	22
2.1.3.3 Gaming . . . . .	22

2.1.3.4	Television and Entertainment . . . . .	22
2.1.3.5	Collaboration . . . . .	23
2.1.3.6	Training . . . . .	23
2.1.3.7	Industrial Design . . . . .	24
2.1.3.8	Archaeology . . . . .	24
2.2	Interest Management . . . . .	24
2.2.1	Scalability of Virtual Environments . . . . .	25
2.2.2	Definition . . . . .	26
2.2.2.1	Interest Management for Scalability . . . . .	26
2.2.2.2	Interest Management for Managing Interests . . . . .	27
2.2.2.3	New Definition . . . . .	28
2.2.3	Techniques for Managing Interests . . . . .	28
2.2.3.1	Class and Value Based Filtering . . . . .	28
2.2.3.2	Domains . . . . .	29
2.2.3.3	Interaction Analysis . . . . .	29
2.2.3.4	Virtual Parallel Worlds . . . . .	30
2.2.3.5	Awareness . . . . .	30
2.2.3.6	Cells . . . . .	31
2.2.3.7	Locales . . . . .	31
2.2.3.8	Visibility Based Filtering . . . . .	31
2.2.4	Interest Management Techniques Categorised . . . . .	32
2.2.4.1	Categorisation . . . . .	32
2.2.4.2	Locales . . . . .	32
2.2.4.3	Interacting Locales . . . . .	33
2.2.4.4	Hybrid Approaches . . . . .	33
2.2.4.5	Technique Mapping . . . . .	33
2.3	Domain Specific Languages . . . . .	35
2.3.1	Readability . . . . .	35
2.3.2	Succinctness . . . . .	36
2.4	Ruby . . . . .	37
2.4.1	ERB . . . . .	38
2.5	Ruby on Rails . . . . .	38
2.5.1	YAML . . . . .	38
2.5.2	RSpec . . . . .	38
2.6	Limitations of current practice . . . . .	38

2.6.1	Assumptions Made . . . . .	39
2.6.1.1	Assumptions of Interest . . . . .	39
2.6.1.2	Assumptions of Capabilities . . . . .	39
2.6.1.3	Implicit Assumptions . . . . .	40
2.6.2	Problems with Assumptions . . . . .	41
2.6.2.1	Assumptions and Interests . . . . .	41
2.6.2.2	Changes in Interest . . . . .	42
2.6.3	Proposed Solution to Interest Management . . . . .	43
<b>3</b>	<b>A Framework for Dynamic Interest Management</b>	<b>44</b>
3.1	A Conceptual Model for Virtual Environments . . . . .	44
3.1.1	Motivations . . . . .	45
3.1.1.1	To Reason About Interest . . . . .	45
3.1.1.2	To be a Model of other Virtual Environments . . . . .	46
3.1.1.3	To be Verifiable . . . . .	46
3.1.2	Axioms . . . . .	46
3.1.2.1	Attributes . . . . .	46
3.1.2.2	Artefacts . . . . .	47
3.1.2.3	Events . . . . .	47
3.1.2.4	Time . . . . .	47
3.1.2.5	Processes . . . . .	48
3.1.3	Users . . . . .	48
3.1.4	Interaction . . . . .	49
3.2	Defining Interest Statements . . . . .	49
3.2.1	An Intensional Definition of Interest . . . . .	50
3.2.2	Interest Conditions . . . . .	51
3.2.3	Combining Interest Conditions . . . . .	52
3.2.4	Auxiliary Sets for Interest Conditions . . . . .	52
3.2.4.1	Derived Sets . . . . .	53
3.2.4.2	Supplementary Sets . . . . .	53
3.2.4.3	An Example: Spatial Sets . . . . .	54
3.2.5	Relative Interests . . . . .	57
3.3	Example Interest Statements . . . . .	58
3.3.1	Locales . . . . .	58
3.3.2	Relative Locales . . . . .	59
3.3.3	Interacting Locales . . . . .	59

3.3.4	Categories . . . . .	59
3.3.5	Combinations . . . . .	60
3.4	Constraints and Conflicts . . . . .	60
3.4.1	Interests for Specific Virtual Environments . . . . .	60
3.4.1.1	Relative Visibility . . . . .	60
3.4.1.2	Constraints . . . . .	61
3.4.1.3	Conflicts . . . . .	63
3.4.2	Separation of Concerns . . . . .	64
3.4.2.1	User Interests: Positive Enforcement . . . . .	64
3.4.2.2	User Interests: Negative Enforcement . . . . .	64
3.4.2.3	Simulation Interests: Positive Enforcement . . . . .	65
3.4.2.4	Simulation Interests: Negative Enforcement . . . . .	65
3.4.3	Combining Interests . . . . .	65
3.5	Summary . . . . .	65
4	<b>Virtual Environment Axioms: A Proof of Concept</b>	<b>67</b>
4.1	The Axioms Revisited . . . . .	67
4.1.1	Artefacts . . . . .	67
4.1.1.1	Artefacts as Objects . . . . .	68
4.1.1.2	Artefacts as Table Rows . . . . .	69
4.1.1.3	Combining Objects and Tables: Object Relational Mapping . . . . .	69
4.1.2	Time . . . . .	70
4.1.3	Events . . . . .	70
4.1.4	Processes . . . . .	71
4.2	Data Design . . . . .	71
4.3	Implementation Decisions . . . . .	72
4.3.1	Data Storage Technology . . . . .	72
4.3.1.1	Persistence . . . . .	72
4.3.1.2	Support for Set Structures . . . . .	73
4.3.1.3	Support for Querying . . . . .	73
4.3.1.4	Support for Spatial Queries . . . . .	73
4.3.1.5	Implementation Choice . . . . .	74
4.3.2	Development Methodology . . . . .	74
4.3.2.1	Test Driven Development . . . . .	74
4.3.2.2	Behaviour Driven Development . . . . .	75
4.3.3	Implementation Language . . . . .	75



4.3.3.1	Supporting Libraries . . . . .	75
4.4	Implementation . . . . .	75
4.4.1	Creating the Database Schema . . . . .	76
4.4.2	Defining the Artefact Class . . . . .	76
4.4.3	A Sample Virtual Environment . . . . .	77
4.4.4	The First Interest Statement . . . . .	78
4.4.5	Viewing the Virtual Environment . . . . .	79
4.4.6	A Client Server Architecture . . . . .	81
4.4.7	An Update Format . . . . .	82
5	Interest Statements . . . . .	85
5.1	Interesting Concepts . . . . .	85
5.1.1	Attributes . . . . .	86
5.1.2	Virtual Attributes . . . . .	86
5.1.3	Relative Artefacts . . . . .	86
5.1.4	Relative Virtual Artefacts . . . . .	87
5.2	Example Statements . . . . .	87
5.2.1	Categories . . . . .	87
5.2.2	Locales . . . . .	87
5.2.3	Relative Locales . . . . .	88
5.2.4	Interacting Locales . . . . .	88
5.2.5	Combinations . . . . .	88
5.3	Representing Interest Statements with SQL . . . . .	89
5.3.1	Categories . . . . .	89
5.3.2	Locales . . . . .	90
5.3.3	Interacting Locales . . . . .	93
5.3.4	Combinations . . . . .	94
5.4	Combining Separate Concerns . . . . .	96
5.5	Limitations . . . . .	97
5.5.1	Expressiveness . . . . .	97
5.5.2	Abstraction . . . . .	97
5.5.3	Readability . . . . .	98
5.5.4	Succinctness . . . . .	98
6	Wish: a DSL for Interest Statements . . . . .	99
6.1	The Structure of a DSL for Interest Statements . . . . .	99

6.1.1	Domain Objectives . . . . .	99
6.1.1.1	Abstraction . . . . .	100
6.1.1.2	Succinctness . . . . .	100
6.1.1.3	Readability . . . . .	101
6.1.1.4	Expressiveness . . . . .	101
6.1.2	Structural Concepts . . . . .	102
6.1.2.1	Interest Conditions . . . . .	102
6.1.2.2	Relative Interest Conditions . . . . .	103
6.1.2.3	Logical Operators . . . . .	103
6.1.2.4	Grouping . . . . .	104
6.1.2.5	Abstraction . . . . .	104
6.1.2.6	Scoping . . . . .	105
6.2	Wish Structure and Syntax . . . . .	105
6.2.1	Interest Conditions . . . . .	106
6.2.1.1	Automagical Value Quoting . . . . .	106
6.2.1.2	Comments . . . . .	107
6.2.2	Relative Interest Conditions . . . . .	108
6.2.3	Logical Operators . . . . .	109
6.2.3.1	not . . . . .	109
6.2.3.2	or . . . . .	109
6.2.3.3	and . . . . .	110
6.2.3.4	and not, or not . . . . .	110
6.2.4	Grouping . . . . .	111
6.2.4.1	Implicit Grouping . . . . .	111
6.2.4.2	Explicit Grouping . . . . .	112
6.2.5	Abstraction . . . . .	113
6.2.5.1	Subwishes . . . . .	113
6.2.5.2	Implicit Parameters . . . . .	114
6.2.5.3	Nested subwishes . . . . .	116
6.2.6	Scoping . . . . .	116
6.2.7	Subwishes: A Myth . . . . .	117
6.3	Design and Implementation . . . . .	117
6.3.1	Agile Development . . . . .	117
6.3.1.1	Specifications . . . . .	117
6.3.1.2	Modularity . . . . .	117

6.3.1.3	Iterative Development . . . . .	118
6.3.1.4	An Example Iteration . . . . .	118
6.3.2	Iterations . . . . .	118
6.3.2.1	Interest Conditions . . . . .	119
6.3.2.2	Explicit Logical Operators: not . . . . .	119
6.3.2.3	Implicit Logical Operators: or, and . . . . .	119
6.3.2.4	Converting a YAML nested list to SQL . . . . .	119
6.3.2.5	Expressions . . . . .	119
6.3.2.6	Auto-quoting . . . . .	119
6.3.2.7	Grouping . . . . .	119
6.3.2.8	Abstraction . . . . .	120
6.3.2.9	Scoping . . . . .	120
6.3.3	Architectural Components . . . . .	120
6.3.3.1	SQL . . . . .	120
6.3.3.2	YAML . . . . .	121
6.3.3.3	Ruby . . . . .	121
6.3.3.4	Wish . . . . .	122
6.3.4	Implementation Overview . . . . .	123
6.3.4.1	YAML to SQL Parser . . . . .	123
6.3.4.2	Ruby erb Evaluation . . . . .	124
6.3.4.3	Wish Auto-quoting . . . . .	126
6.3.4.4	Wish Grouping . . . . .	126
6.3.4.5	Wish Abstraction . . . . .	128
6.3.4.6	Wish Scoping . . . . .	128
6.3.4.7	Overview . . . . .	129
<b>7</b>	<b>Case Study and Evaluation</b>	<b>131</b>
7.1	Case Study . . . . .	131
7.1.1	Artefacts . . . . .	131
7.1.2	Football Pitch . . . . .	132
7.1.3	Players . . . . .	134
7.1.4	Referee . . . . .	136
7.1.5	Football . . . . .	137
7.1.6	Locales . . . . .	137
7.1.7	Auras . . . . .	139
7.1.8	All Artefacts . . . . .	139

<b>7.2 Example Statements</b>	<b>139</b>
7.2.1 Categories	143
7.2.2 Locales	143
7.2.3 Relative Locales	145
7.2.4 Interacting Locales	146
7.2.5 Combinations	147
7.2.6 Combining Concerns	149
<b>7.3 Dynamic Interests</b>	<b>149</b>
<b>7.4 Evaluating The Domain Objectives</b>	<b>153</b>
7.4.1 Abstraction	154
7.4.2 Readability	154
7.4.2.1 Subwish Names	154
7.4.2.2 Visual Structure	155
7.4.2.3 Abstracting Complexity	155
7.4.2.4 Removing Ambiguity	156
7.4.3 Succinctness	157
7.4.4 Expressiveness	157
<b>8 Conclusions and Further Work</b>	<b>158</b>
8.1 Summaries	158
8.1.1 Contribution Summary	158
8.1.2 Chapter Summary	159
8.2 Final Thoughts	160
8.2.1 Technology Choices	160
8.2.2 Treating Wish as an Essay	161
8.2.3 Component Objectives	162
8.3 Further Work	163
8.3.1 Wish as a DSL for Information Scoping	164
8.3.2 Separating Relationships from the Data	164
8.3.3 Resource Costs	165
8.3.4 Rich Interest Conditions	165
8.3.5 Compiling Wish to Other Representations	165
8.3.6 Interesting Events	166
8.3.7 Prioritised Events	166

<b>A Example Iteration</b>	<b>167</b>
A.1 Wish Auto-quoting Implementation . . . . .	167
A.2 Auto-quoting Specification Output . . . . .	168
A.3 Example RSpec Specification: Auto-quoting . . . . .	169
<b>B Case Study Data</b>	<b>176</b>
B.1 Football Pitch . . . . .	176
B.2 Players . . . . .	177
B.2.1 Red Team . . . . .	177
B.2.2 Blue Team . . . . .	177
B.2.3 Goalkeepers . . . . .	177
B.2.4 Referee . . . . .	178
B.3 Football . . . . .	178
B.4 Locales . . . . .	178
B.5 Auras . . . . .	178
B.6 All Artefacts . . . . .	178
<b>C Case Study Example Statements</b>	<b>182</b>
C.1 Relative Artefacts . . . . .	182
C.2 Categories . . . . .	182
C.2.1 English Prose . . . . .	182
C.2.2 Wish . . . . .	182
C.2.3 SQL . . . . .	182
C.2.4 Matching Artefacts . . . . .	182
C.3 Locales . . . . .	182
C.3.1 English Prose . . . . .	182
C.3.2 Wish . . . . .	184
C.3.3 SQL . . . . .	184
C.3.4 Matching Artefacts . . . . .	184
C.4 Relative Locales . . . . .	185
C.4.1 English Prose . . . . .	185
C.4.2 Wish . . . . .	185
C.4.3 SQL . . . . .	185
C.4.4 Matching Artefacts . . . . .	185
C.5 Interacting Locales . . . . .	185
C.5.1 English Prose . . . . .	185

C.5.2	Wish . . . . .	185
C.5.3	SQL . . . . .	186
C.5.4	Matching Artefacts . . . . .	186
C.6	Combinations . . . . .	186
C.6.1	English Prose . . . . .	186
C.6.2	Wish . . . . .	186
C.6.3	SQL . . . . .	186
C.6.4	Matching Artefacts . . . . .	187
<b>D</b>	<b>Sub Wishes</b>	<b>188</b>
D.1	In Awareness Range Of . . . . .	188
D.2	Auras In awareness Range Of . . . . .	188
D.3	Overlaps . . . . .	188
D.4	Coloured . . . . .	188
D.5	Named . . . . .	188
D.6	Is . . . . .	189
D.7	Within Circle . . . . .	189
D.8	Within Box . . . . .	189
D.9	Within Cube . . . . .	189
D.10	Near To . . . . .	189
D.11	Virtual . . . . .	190
D.12	Categorised As . . . . .	190
<b>Bibliography</b>		<b>191</b>

# List of Tables

2.1	Categorisation of Interest Management Techniques . . . . .	34
3.1	Example Interest Conditions . . . . .	51
3.2	The Standard Logical Operators . . . . .	52
3.3	Combining Interest Statements with Logical Operators . . . . .	52
3.4	Derived Sets . . . . .	53
3.5	The Standard Set Operators . . . . .	54
3.6	The Mapping between Condition Combinations and Derived Sets . . . . .	54
3.7	Spatial Operators . . . . .	56
3.8	Useful Spatial Sets . . . . .	58
3.9	Dealing with Constraints . . . . .	62
3.10	Combining Concerns of Interest . . . . .	65
4.1	Mapping between Conceptual Model Term, Database Term, and Object-Oriented Term . .	72
4.2	Attributes for the Initial Design . . . . .	73
5.1	Deconstructing and Evaluating a Category Based Set Builder . . . . .	90
5.2	SQL Conditional Operators . . . . .	90
6.1	Logical Combinational Operators . . . . .	103
6.2	Value Quoting Examples . . . . .	107
6.3	Evaluating Interest Condition Expressions . . . . .	109
7.1	Artefact Attributes . . . . .	132
7.2	Football Pitch Artefacts . . . . .	133
7.3	All Red Players . . . . .	134
7.4	All Blue Players . . . . .	135
7.5	Goalkeepers . . . . .	136
7.6	The Referee . . . . .	136
7.7	The Football . . . . .	137
7.8	Locales . . . . .	137

7.9	Auras . . . . .	139
7.10	All Artefacts . . . . .	141
7.11	All Artefacts (Continued from Table B.9) . . . . .	142
7.12	Turquoise Artefacts . . . . .	150
7.13	Blue Artefacts . . . . .	152
7.14	Artefacts within the Home Penalty Circle . . . . .	154
B.1	Football Pitch Artefacts . . . . .	176
B.2	All Red Players . . . . .	177
B.3	All Blue Players . . . . .	177
B.4	Goalkeepers . . . . .	177
B.5	The Referee . . . . .	178
B.6	The Football . . . . .	178
B.7	Locales . . . . .	178
B.8	Auras . . . . .	179
B.9	All Artefacts . . . . .	180
B.10	All Artefacts (Continued from Table B.9) . . . . .	181
C.1	Red Artefacts . . . . .	183
C.2	Artefacts on the Near Side of the Football Pitch . . . . .	184
C.3	Football Pitch Artefacts . . . . .	185
C.4	Artefacts Matching the Interacting Locales Example . . . . .	186
C.5	Artefacts Matching the Combinations Example . . . . .	187



# List of Figures

1.1	Number of Active Subscriptions for MMOGs from Jan 1997 to July 2006[121]. . . . .	6
1.2	Scenario Overview . . . . .	13
1.3	Low Resolution Image of the Patient . . . . .	13
1.4	High Resolution Image of a Heart . . . . .	14
3.1	An External Entity or User . . . . .	48
3.2	Interesting Artefacts . . . . .	50
3.3	Interesting Artefacts . . . . .	51
3.4	More Interesting Artefacts . . . . .	51
3.5	Spatial Sets within a Virtual Environment . . . . .	55
3.6	spatial Sets can be of any shape or orientation . . . . .	57
3.7	Relative Visibility . . . . .	61
3.8	Constraints Introduced . . . . .	61
3.9	Updated Interests for Users <i>A</i> and <i>B</i> . . . . .	62
3.10	Blocked Visibility . . . . .	63
4.1	The Architecture of the Viewing System . . . . .	80
4.2	A Simple World Consisting of a Cuboid, Cylinder, Sphere, and Floor . . . . .	81
4.3	A Client Server Architecture . . . . .	81
5.1	A Simple Representation of a Football Pitch . . . . .	91
5.2	Determining whether a Given Artefact's x and y Coordinates Fall Within the Area of a Football Pitch . . . . .	92
5.3	Determining whether Two Circles Overlap . . . . .	93
5.4	Determining whether a Given Point is Within a Circle . . . . .	95
6.1	The Wish Components . . . . .	120
6.2	Converting a Wish Statement to SQL . . . . .	121
6.3	Parsing a YAML Statement and Converting it to SQL . . . . .	121
6.4	Parsing a YAML + erb Statement and Converting it to SQL . . . . .	122

6.5	Parsing a Wish Statement and Converting it to SQL . . . . .	122
6.6	Converting a YAML String Element to SQL . . . . .	123
6.7	Combining YAML Elements . . . . .	124
6.8	Converting YAML to SQL . . . . .	125
6.9	The Wish Auto-Quoting Algorithm . . . . .	127
6.10	The Wish Compiler . . . . .	130
7.1	Aerial View of the Football Pitch . . . . .	132
7.2	Stadium View of the Football Pitch . . . . .	133
7.3	Football Pitch Areas . . . . .	134
7.4	The Red Team . . . . .	135
7.5	The Blue Team . . . . .	135
7.6	The Goalkeepers . . . . .	136
7.7	The Referee . . . . .	137
7.8	The Football . . . . .	138
7.9	The Locale Representing the Near Half of the Pitch . . . . .	138
7.10	All the Auras . . . . .	140
7.11	A Stadium View of All Artefacts . . . . .	140
7.12	An Aerial View of All Artefacts . . . . .	142
7.13	All Red Artefacts . . . . .	143
7.14	All Artefacts Within the Near Half of the Pitch . . . . .	144
7.15	Artefacts Within the Referee's Aura . . . . .	146
7.16	All Artefacts . . . . .	147
7.17	All Artefacts that are in Awareness Range of the Referee . . . . .	148
7.18	Results of a Combination Statement . . . . .	148
8.1	Spatial Map Indicating the Objectives of Wish Components . . . . .	163

# Chapter 1

## Introduction

The concepts involved in designing, implementing and using virtual environments are both widely understood and practised. However, there are still some limitations of these environments which hinder their potential for scalability and, in particular, adaptability. This thesis focusses on the representation of interests within virtual environments. The techniques introduced are based upon the concept of dynamic interest management. They de-couple the logic that implements the interest management from the virtual environment's implementation. This allows users to influence what he or she is interested in, and subsequently change their interests.

As a general introduction to this work, this chapter aims to describe in detail detail the motivations of this thesis, in order to place the focus and goals in a wider perspective and context. Section 1.1 introduces scalability and adaptability as two of the remaining challenges of virtual environments. It explains the motivations behind this thesis with respect to these challenges. Section 1.2 introduces a mathematical model that illustrates the relationship between messages and interest management. Sections 1.3 and 1.4 discuss various approaches to tackling the issues of scalability and adaptability respectively. Section 1.5 then argues the case for dynamic interest management, and explains how it can be used to improve both scalability and adaptability. This is further illustrated by a scenario presented in Section 1.6, which is discussed in Section 1.6.1. Finally, Sections 1.7 and 1.8 describe the contributions and structure of this thesis respectively.

### 1.1 Limitations in Virtual Environments

Virtual environments<sup>1</sup> are computer generated environments that can be used for a broad set of activities including simulation, visualisation and collaboration<sup>2</sup>. Although there has been over 20 years of research into the area, particularly within academia and the gaming industry[95], there are still some serious computing science challenges remaining. These include scalability and adaptability. Sections 1.1.1 and 1.1.2 discuss these issues in greater detail.

---

<sup>1</sup>Virtual environments are defined in greater detail in Section 2.1.1

<sup>2</sup>The range of uses of virtual environments is discussed further in Section 2.1.3

### 1.1.1 Scalability

Scalability has long been seen as a major challenge for virtual environments. The development of war simulations and larger and larger massive multiplayer online games, MMOGs, has required an increase in the size and complexity of these environments. This trend is clear when we consider commercial online multi-player games such as Quake[54], Second Life[58] and World of Warcraft[57]. Some of the first games in this genre had severe limitations in terms of the potential number of simultaneous users. For example, in 1996 the game Quake created by ID Software[54] only supported 16 simultaneous players. However, if we consider modern products in this genre, such as Blizzard's World of Warcraft[57], we see offerings able to support over 6,500,000 current subscribers (see Figure 1.1), and hundreds of thousands of concurrent players[24]. Unfortunately, as the number of participants grows, the contention on shared resources becomes more severe. However, various algorithms and techniques have been created to achieve this increase in scalability, as discussed in Chapter 2.

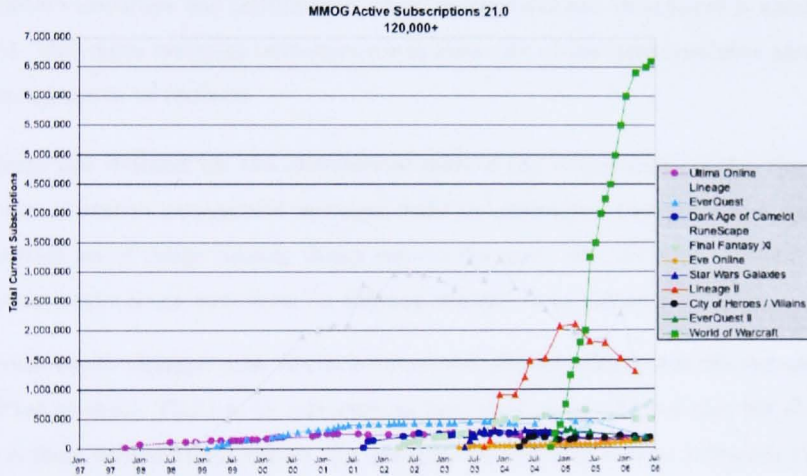


Figure 1.1: Number of Active Subscriptions for MMOGs from Jan 1997 to July 2006[121].

Everything that can happen in a virtual environment requires either processing power or communication with entities within or outside the system. This thesis is not concerned with reducing the processing power necessary for the virtual environment to exist. There are currently many successful approaches which allow a virtual environment implementation to be clustered over many servers working together. However, network resources still remain very expensive, particularly compared with computational resources[48]. This thesis is concerned with techniques capable of helping to manage these network resources, particularly through the ability to tightly control the communication between the virtual environment and its users. One of the motivations for this might be the constraints introduced by a limited bandwidth. Having a fine grained level of control of the data communicated is a very important issue for

clients using connections with a very low bandwidth, or who have to pay-per-byte. It also affects servers with a growing number of connected users, as the number of messages the server needs to send increases quadratically in relation to the number of users<sup>3</sup>.

### 1.1.2 Adaptability

The adaptability of a virtual environment is another way of describing its flexibility or its ability to cope with changing circumstances. For example, the capabilities of the system may change over time, the number of people using and interacting with the system may change, and the tasks that may be performed in the system may also change. In response to this, the system may adapt in various ways such as prioritising certain activities, optimising any computation, or reducing the frequency of communication.

Assumptions generally have to be made in order to optimise systems. Unfortunately the more assumptions made, the less flexible the system becomes. One of the goals of adaptability is to not make any unnecessary assumptions, and allow the system to be as adaptable as possible whilst maintaining both the system's usefulness and performance. These assumptions will be explored in greater detail in Section 2.6.1. This thesis evaluates techniques which allow two of the most restrictive assumptions in virtual environments to be removed:

- **Interests are defined by the simulation:** most of the virtual environments that implement some sort of interest management technique make the assumption that the user is only interested in a certain set of things (usually things near to the user). The ability to represent changes in individual tactics is one motivation for allowing interests to be influenced by users.
- **Interests never change:** most virtual environments also assume that the user never changes his or her interests. This can be restrictive as individual user's interests may not always be the same as those defined by the system. For example, the user may want to arbitrarily change his or her tactics.

## 1.2 A Mathematical Model of the Relationship between Number of Messages and Users within a Virtual Environment

Section 1.1.1 introduced the issue of scalability. It described the trend in virtual environments for increasing the potential number of simultaneous users. This section introduces a mathematical model of the number of messages needed in a simple virtual environment, and explains that as the number of users increases, the number of messages that the system needs to generate and transmit can quickly exceed any bandwidth limitations.

---

<sup>3</sup>See Section 1.2 for a further discussion of this relationship.

As described in detail in Section 3.1, a virtual environment can be seen to be composed of artefacts that are affected by events. For the users of a virtual environment to be made aware of an event, they must be sent a message describing it. In this simple model, the only artefacts in the virtual environment are those representing a user<sup>4</sup>.

Consider the following assumptions:

- i) The total number of users in a system is  $N$ .
- ii) Each user creates  $e$  events per unit of time.
- iii)  $\mu$  is the average size of a message.
- iv) Each user needs to be made aware of all events.

The total number of events ( $E$ ) that are generated per unit of time can be calculated as:

$$E = Ne \quad (1.1)$$

Following assumption iv, the number of messages ( $M$ ), per unit of time, that needs to be sent is:

$$M = EN = N^2e \quad (1.2)$$

The bandwidth ( $b$ ) needed to communicate this information, for a unit of time, can be calculated as:

$$b = M\mu \quad (1.3)$$

or:

$$b = N^2e\mu \quad (1.4)$$

or:

$$b = kN^2 \quad (1.5)$$

(where  $k$  is a constant).

Therefore, as we increase the number of users in the system, the number of messages that the system needs to send, and the bandwidth required, increases quadratically, and the contention on shared resources becomes more severe[83]. Extra messages sent represent a cost in terms of network bandwidth, routers buffer occupation and end host resources, augmenting latency[67][5]. This is, of course, assuming that each user is made aware of all events.

---

<sup>4</sup>Artefacts representing users are typically called avatars. See Section 3.1.3 for a further discussion.

## 1.3 Approaches to Scalability

A virtual environment can be seen as consisting of two parts: application logic, and the execution environment. The application logic is software which defines the processes that the environment is constructed from. The execution environment is the system(s)<sup>5</sup> that executes those processes, stores the application state, and provides communication with other systems (e.g. clients). In attempting to increase the scalability of a virtual environment, we can approach these two parts separately. We can improve the system resources and efficiency, and also the efficiency of the application itself. The following sections, 1.3.1 and 1.3.2, explore both approaches.

### 1.3.1 Improving System Resources and Efficiency

This approach attempts to increase efficiency through improving the hardware capability, protocol efficiency, and overall design of the system itself. In this context, design refers to the structures within which different hardware components are linked together.

Of course, faster and better hardware can always be purchased. Unfortunately, as we tend toward the limitations of current technology we find that the speed-up achieved by combining resources in a parallel fashion is rarely linear. Communication overheads between computational nodes, and the varying efficiencies of parallel algorithms for certain types of problems, reduce the potential for linear speed-up. Regardless of this, a linear speed-up is not a general solution to a quadratic problem.

Noticing that many of the messages sent throughout the system are identical, networking protocols and smart routers may be employed to solve the problem of unnecessary duplication. In fact protocols for this problem such as multicast already exist[35], and there are even approaches to optimise the explicit use of multicast groups[5]. Unfortunately support for protocols like multicast is not pervasive in today's Internet infrastructure, and therefore it is not currently safe to rely upon them to build scalable virtual environments for the average consumer. Also, reducing the number of messages that the server has to send does not reduce the volume of messages that the clients have to receive.

Considering the example given in Section 1.2, we can see that this approach to scalability attempts to reduce the size of messages ( $\mu$ ), and also potentially reduce the bandwidth required ( $b$ ) on the server by using techniques such as multicast in order to not actually have to send  $M$  (defined in Equation 1.2) messages, for  $M$  messages to be received across all the clients. However if we consider that our clients have bandwidth constraints, then the scalability of our system is restrained by the number of messages received across all clients. Rather than attempt to optimise the network usage, it may be more feasible to reduce the amount of information that is received by each individual client. This, of course, maps on to a reduction of messages that have to be sent by the server.

---

<sup>5</sup>The execution environment is typically a combination of both hardware and software.

Regardless of the system resources available, there will always be a limitation on the number of messages that can be generated and sent per unit of time. There will also be a limit on the funds available to purchase the resources. Network resources also still remain very expensive compared with computational resources[48]. There is therefore a motivation to increase the efficiency of any system. Improving the system resources and efficiency can only get us part of the way to a solution to the issue of scalability. This is clearly evident when we consider that there may be a range of clients interacting with the server, and upgrading them all simultaneously would be a non-trivial task.

### 1.3.2 Increasing Application Efficiency

A top-down approach to the issues of scalability would challenge the overall design of the application. It would free us to challenge the following assumption from Section 1.2:

iv) Let us also assume that each user needs to be made aware of all events.

For it is challenging this assumption that allows the following inference to also be challenged:

*“as we increase the number of users in the system, the number of messages that the system needs to generate and send, and the bandwidth required, increases quadratically.”*

If it is assumed that all users do not need to be made aware of all events, it is therefore necessary to create rules for each user, dividing the set of all events into two sub-sets: events to send, and events to discard. This technique is called interest management.

Again, referring to the example in Section 1.2, we see that the removal of assumption iv changes the number of messages,  $M$ . Instead of simply being the number of events generated multiplied by the number of users as in equation 1.2, each user now only receives a subset of all the events:

$$\sigma \in [0, 1] \quad (1.6)$$

$$\sigma EN \quad (1.7)$$

where  $\sigma$  is the ratio of artefacts that are interesting and is inversely proportional to the size of the subset of events each user receives<sup>6</sup>. This has the property of changing our equation for calculating bandwidth (b) to this:

$$b = \sigma k N^2 \quad (1.8)$$

Note that  $b$  still remains quadratic in  $N$

$$f(x) = kx^2 \quad (1.9)$$

---

<sup>6</sup>Assuming that the proportion of events that are interesting for each user is constant.



However, by having the ability to modify the value of  $\sigma$  we gain an element of control over the gradient of the curve; reducing both the number of messages and total amount of bandwidth needed. This would therefore allow us to cater for potentially many more people for the same value of bandwidth,  $b$ .

## 1.4 Approaches to Adaptability

Typically, an increase in scalability comes at a price: that of adaptability. In order to achieve a high level of scalability, many assumptions have to be engineered into the systems. This often results in a massive reduction in their flexibility. No matter how many rules are inserted into systems we cannot ever predict the behaviour of its users. This is especially true over a long period of time, and when the user base is very large and diverse. If the longevity of a virtual environment is a requirement, then it is desirable to increase the flexibility of these systems, and therefore increase the ability to adapt to change. Examples of assumptions, and systems that contain assumptions are introduced in Section 2.6.1.

Interest management is a key technique to address the scalability problem of virtual environments. As discussed in Section 2.2.2.1, it has been used almost exclusively to tackle issues of scalability, network bandwidth and computer processing power. However, as discussed in Section 2.2.2.2, it can also be used as a valid technique for increasing a system's adaptability. Although the goals of scalability and adaptability are often orthogonal, they share the ability to be able to benefit from the introduction of interest management techniques.

## 1.5 Dynamic Interest Management for Adaptable Virtual Environments

As stated in Section 1.3.2, interest management is a key part of tackling the issue of scalability in virtual environments. One of the major problems of interest management techniques is their inherently static nature. Interest management is a set of assumptions of interest used to reduce a virtual environment's workload. Interest management can be a very effective mechanism for reducing resource usage like network bandwidth and computer processing power[33] by limiting the amount of information that must be processed by each user. As discussed in Section 1.4, assumptions made can reduce adaptability. It can also be difficult to predict interests. This is especially true if we consider that the context within which the original assumptions were made is likely to change. By de-coupling the logic that implements the interest management from the main virtual environment logic, this thesis proposes a technique that allows interests to be changed before and during the execution of the environment.

In addition to it being potentially unsafe to assume that interests within a virtual environment may never change, it is also potentially unsafe to assume that the characteristics of the virtual environment

itself may never change. The following are examples of characteristics which may change during the life-time of a virtual environment:

- the underlying hardware of the virtual environment,
- the hardware capabilities of clients,
- the available bandwidth,
- the load<sup>7</sup>,
- the rules within the virtual environment (e.g. game/tactic/simulation type changes).

Adaptability provides a powerful way of approaching some of the problems virtual environments face. For example, adaptability makes it possible to react to an overloaded system by reducing the scope of interest (hence reducing the number of messages needed to be computed and sent).

Essentially this removes the initial assumption that the value of  $\sigma$  is the same for each client, or in other words: *“the proportion of events that are interesting for each user is constant”*. By allowing each client to influence their value of  $\sigma$ , their ability to adapt increases. It also allows users to customise systems to their interests<sup>8</sup>.

## 1.6 A Scenario

The following scenario illustrates some of the advantages of having an adaptable virtual environment.

A surgeon is about to perform a heart operation on a patient using a new advanced procedure which involves the surgeon and the patient being located in different places. This is achieved using a virtual representation of the patient’s body generated by sensors in the operating theatre<sup>9</sup>. The actual physical operation is performed by a machine. This is controlled by the surgeon who interacts with a local client, that provides a haptic feedback device for input, and a visualisation of the virtual representation of the operation as the output.

Figure 1.2 shows an overview of this situation. As the bandwidth between the virtual representation and the client is limited, it is important to optimise its usage for the particular situation that it is representing. For example, before the operation starts the surgeon may want to examine many of the sensor’s readings to get an overall view of the patient’s status.

The surgeon chooses to wait for all of the sensor readings to become visible on his client, including a low resolution graphical representation of the patient (see Figure 1.3). Satisfied that all is ready to go

---

<sup>7</sup>The load is potentially proportional to the size and complexity of the world.

<sup>8</sup>For examples of methods with which to express interests see Section 2.2.3.

<sup>9</sup>A similar approach has already been used to teach operational procedures[103].

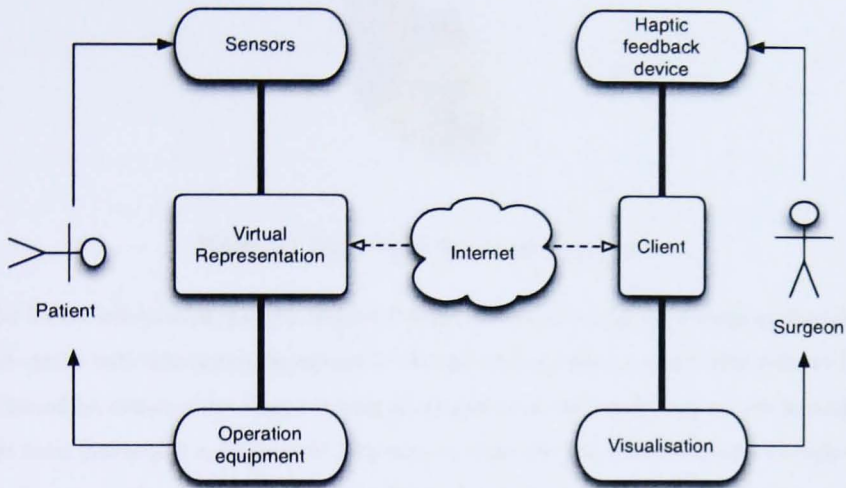


Figure 1.2: Scenario Overview

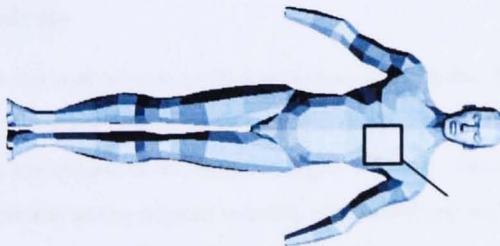


Figure 1.3: Low Resolution Image of the Patient

ahead, he indicates to the system that he wishes to have a high resolution graphical representation of only the patient's heart (see Figure 1.4) to analyse what he has to do, the operation then starts.



Figure 1.4: High Resolution Image of a Heart

In order for the information that the surgeon's client retrieves from the virtual representation to be as useful as it can be only information important to the operation is sent i.e. events that refer to the heart. This is achieved by reducing the surgeon's area of interest from the whole body to the immediate area around the heart (see area A in Figure 1.3). The surgeon then tells the client the types of artefact that he is interested in i.e. the heart, veins and arteries. The virtual representation then only sends information regarding these artefacts - thus artefacts that previously obscured the heart (such as the patient's skin and rib-cage) are no longer visible - allowing the surgeon to concentrate his work on the vital artefacts of the operation.

During the operation the patient starts to display signs of increased sensitivity. These signs are visible on the patient's face. As these events are outside of the surgeon's chosen interest they are not visible to him. However, the system providing the virtual representation decides that this information is crucial for the surgeon to be aware of and sends it to him. The surgeon sees this new information and is able to resolve the situation by applying more anaesthetic.

### 1.6.1 Scenario analysis

In the scenario the virtual representation is, in fact, a virtual environment. The surgeon interacts within this virtual environment by creating events. These events alter the state of the virtual environment which may result in a change in the output of the client's output device(s). The scenario also introduces the concept of interest management as the surgeon controls which artefacts and events within the world he receives, (in essence culling the set of all world artefacts and events to a new set which only contains interesting entities). In this case, this is achieved using the following two techniques:

- **Locale:** the surgeon specified a particular area of interest (in this case area A of Figure 1.3).
- **Categorisation:** the surgeon specified a set of classifications of artefacts that he was interested in.

Notice that in the scenario, both the surgeon and the simulation itself specified their interests while the virtual environment was executing.

## 1.7 Contributions

This thesis makes the following contributions to research into interest management within virtual environments:

**Taxonomy of currently used interest management techniques.** The various techniques used for interest management are surveyed and discussed. Categorisation, locales and interacting locales are introduced as three general techniques, and it is shown how the various surveyed techniques can be mapped on to them.

**A conceptual model of interests based on set-theory.** The taxonomy of interest management techniques is formalised using set-theory, and then implemented using SQL as a proof of concept.

**Wish, a domain specific language for representing interests.** The implementation of the formalisation of the interest management techniques is critiqued, and shown to have limitations in its usefulness such as a lack of readability, succinctness and no ability to allow for abstractions. These limitations are overcome through the design of a new domain specific language. This new language, Wish, is then evaluated using a case study.

## 1.8 Thesis structure

Chapter 2 presents background information, introducing the concept of a virtual environment in its most abstract sense. Different conceptual models are introduced, described and analysed. The motivations for Interest Management techniques are introduced followed by a survey of the available techniques used in current virtual environment implementations. Chapter 3 introduces the novel categorisation conceptual model for virtual environments and describes how it can be used to implement dynamic interest management, and to represent various interest management techniques as introduced in Chapter 2. Chapter 4 describes an implementation of the virtual environment axioms. Chapter 5 uses SQL to build an implementation of the categorisation conceptual model introduced in Chapter 3. Chapter 6 focusses on the limitations of using SQL as the implementation language, and introduces Wish, a novel domain specific language which, compared to the SQL equivalent, is more readable, succinct and has support for abstraction. Chapter 7 evaluates Wish using a case study, and finally Chapter 8 concludes this thesis.

## Chapter 2

# Literature Survey

Virtual environments have been investigated for many years, with contributions ranging from simple simulations to massive commercial multiplayer games. There has been an enormous research focus on the scalability of these environments, with various techniques proposed in order to meet this challenge such as data distribution management, and interest management.

This chapter aims to provide the reader with a brief introduction to virtual environments, leading on to a discussion of some of the major challenges faced by the field. The chapter will introduce the various techniques used to meet the issue of scalability, and will finally focus on the technique of interest management. The various concepts within, and implementations of, interest management will be surveyed. The chapter then introduces the key technologies used in this thesis such as Ruby, YAML, and domain specific languages. Finally the chapter will conclude with a discussion of some of the limitations of current virtual environment interest management techniques.

## 2.1 Virtual Environments

This section will introduce the concept of a virtual environment. Starting with a study of alternative definitions, the section will then explore the origins of virtual environments, and finally illustrate the range of usage of virtual environment systems.

### 2.1.1 Definition

This section will introduce the broad range of definitions for the term virtual environment that are available in the literature. It will then introduce its own definition which will be used throughout the rest of this thesis<sup>1</sup>.

---

<sup>1</sup>It is important to note that these definitions are purely for the purpose of providing an agreed notion of the concepts discussed for the rest of the thesis to build upon.

### 2.1.1.1 Range of Definitions

Unfortunately for the task of determining the origins of virtual environments, there is no real consensus on what actually constitutes a virtual environment. Stuart[107] agrees, and says: *“There are a great many definitions offered by different researchers”*. However, he turns this point on its head suggesting that *“nearly everyone agrees that certain current systems provide virtual environments”*. The range of these *certain systems* is introduced in Section 2.1.3.

So, what is the range of definitions that Stuart describes? Well we can start with one of his own:

*“A VE is an interactive, immersive, multisensory, 3D synthetic environment”*

He breaks this definition down as follows:

- By immersive, I mean that rather than looking at and listening to a display coming from a typical small computer monitor, the display creates the impression that you’re inside the environment produced by the computer.
- By multi-sensory, I mean that more than one sensory modality is used to display the environment visual, auditory, haptic, etc.
- By 3D, I mean that not only does the environment appear to the user to surround him, but cues are also given to convey that it has depth and the user can move through it.
- By synthetic, I mean that the environment is generated by a computer system (it is not, for example, pre-recorded)[107].

Through this definition, Stuart actually describes the kind of systems introduced in Section 2.1.3. However, it can be argued that a virtual environment does not necessarily have to be immersive, multi-sensory or 3D[38]. For example, the MASSIVE-1 system featured a text only interface[44]. These are properties that a virtual environment could offer, but are not necessary for its existence and usefulness. Consider the case of immersion. It may be possible to experience a virtual environment externally, rather than from an immersed perspective. It may also be argued that the synthetic nature of a virtual environment is driven by users immersed in a multi-sensory 3D environment. However, it is possible, and often interesting, to gather results from the real world (i.e. placement of people in a city) and use this data to drive the environment. This highlights the point that a virtual environment does not necessarily have to even contain users. I therefore only agree with the final sentence - that a virtual environment must be synthetic. I feel that the other parts of his definition (immersive, multi-sensory, and 3D) are interesting and useful properties of many virtual environments, but not general properties that define a virtual environment.

Consider another definition:

*“A system through which users may interact with each other and collaborate through a virtual synthetic world”*(Oliveira[31])

In this definition we see the previously used words *interact* and *synthetic*. We are also introduced to the concept of collaboration between users of this virtual synthetic world. Collaboration and in particular the term Collaborative Virtual Environment (CVE) is introduced as a particular type of virtual environment in Section 2.1.1.2. There are two problems here which are unique to this definition. The first is the usage of the term collaborate. There are many virtual environments that do not facilitate collaboration. For example, 3D renderings of architectural plans or any standard VRML world[117]. The second problem is the use of the word virtual. A definition should not contain the words or terms it is itself defining. The same is true of the following definition:

*“virtual environment. An environment which is partially or totally based on computer generated sensory inputs.”*(Federation of American Scientists[90])

Looking further, both wikipedia and answers.com redirect you to their definition of virtual reality.

*“Virtual reality (VR) is a technology which allows a user to interact with a computer-simulated environment.”*(Wikipedia[120])

Unfortunately this definition only goes half-way to helping us define a virtual environment, as it uses the word environment, which is half of the term we’re trying to define.

#### 2.1.1.2 Use of Modifiers

If we look at some of the terms that were rejected in the process of attempting to define the term virtual environment in Section 2.1.1, we are given an insight to the range of usage of virtual environments. These terms include the following modifiers: interaction, multi-sensory, collaboration, 3D and synthetic. There are other terms too e.g. multi-user, networked, large-scale, distributed. These terms describe the various flavours of virtual environments in use, and of course they come in acronym form. Consider the following example variants:

- **VE** - virtual environments[16],
- **CVE** - collaborative virtual environments[25],
- **NVE** - networked virtual environments[75],
- **DVE** - distributed virtual environments[80],
- **LCVE** - large-scale collaborative virtual environments[68],
- **MUDVE** - multi-user distributed virtual environment[83],



- **LSVE** - large scale virtual environments[66]
- **LDVE** - large scale distributed virtual environments[17],
- **VW** - virtual worlds[98].

Matijasevic[78] also discusses this issue suggesting that there are many flavours of VR terms citing *augmented reality*, *artificial reality*, and *synthetic environment* as some of them. There is also some conflation with the discipline computer supported cooperative work (CSCW)[47]. Bartlett[8] also agrees:

*“One of the largest problems facing current DVE development is a lack of order: even the topic area Distributed Virtual Environments possesses numerous synonyms; e.g. Networked Virtual Environment (NVE), Collaborative Virtual Environment (CVE) and to a lesser extent, Computer Supported Collaborative Work (CSCW).”*

What we are seeing here is a combination of two things: an overloading of terms, and the creation of new descriptive terms to represent specific focuses or properties of particular virtual environments. Unfortunately, the net effect of this is the pollution of the virtual environment namespace, resulting in very many terms all describing potentially very small differences. For example, consider the difference between the terms networked virtual environment and distributed virtual environment. A far worse result of this situation is where terms start to become implicit. We therefore have some people being explicit with their terms (MUDVE being a good example) yet others using the term VE, and assuming people understand that they are implicitly referring to a collaborative, multi-user, distributed virtual environment. The problem is that the term virtual environment can be as broad as a term such as programming language.

### 2.1.1.3 New Definition

In an absence of a clear agreed consensus on a definition, let us attempt to create our own for the purposes of this thesis. Let us use a literal definition of the component words as a starting point:

**vir·tu·al** ˌvɜːtʃuəl

adjective

almost or nearly as described, but not completely or according to strict definition [82]

This definition describes the notion of a concept being very similar to, but not exactly the same as, another concept. This could refer to both imitation or emulation. Let us also consider the following definition:

**virtual**, **a.** (and **n.**)

1. **a.** Possessed of certain physical virtues or capacities; effective in respect of inherent natural qualities or powers; capable of exerting influence by means of such qualities. [61]

This definition, taken from the Oxford English Dictionary, refers to entities that have virtues, or attributes that are capable of exerting influence or interacting with our world through such attributes. This loosely ties in with the previous definition by referring to imitations or emulations of real world entities that are able to influence the real world. The concept of software fulfils this criteria, being constructed with language, yet capable of printing results, drawing pictures, producing sounds, moving arms, etc.<sup>2</sup> This brings us back to the Oxford English Dictionary and a definition within the scope of computing:

*virtual, a. (and n.)*

*g. Computers.* Not physically existing as such but made by software to appear to do so from the point of view of the program or the user; *spec.* applied to memory that appears to be internal although most of it is external, transfer between the two being made automatically as required. [61]

Looking at this computing related definition, we can infer that a virtual environment is a software entity which intends to emulate or imitate something. The *spec* part of the definition refers to a specific example of this. It describes the concept of virtual memory whereby the operating system emulates physical RAM using the capacity of an attached hard-drive.

The environment component of the term virtual environment alludes to the concept that is to be emulated or imitated. Let us also look at some definitions of that term:

*environment*

*2. concr. a.* That which environs; the objects or the region surrounding anything.[61]

*environment, n.*

Add: [2.] *e. Computing.* The overall physical, systematic, or logical structure within which (a part of) a computer or program can operate; the particular combination of operating system, software tools, interface, etc., through which a user operates or programs a system.[61]

Again, we have two definitions: a generic one, and one specific to computing. The generic one introduces three important concepts: object, region and the notion of enclosing or containing. The environment is therefore a container, region or set of objects which surround other items. Herein lies an interesting philosophical question: what are those items that are surrounded by an environment? For this I will offer a recursive answer and suggest that those items can be an environment, region or object.

The computing definition acts as a red herring in our quest to define the term virtual environment, and really defines the term computing environment rather than the more general term, environment.

<sup>2</sup> "The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures. Yet the program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. It prints results, draws pictures, produces sounds, moves arms." [19]

It describes the whole computing system which is presented to a user. An implementation of a virtual environment (software and hardware) would itself be an environment in this sense, but it's a confusing overloading of the term.

Pulling the relevant parts of all these definitions together, I define the term virtual environment as the following<sup>3</sup>:

*A region, constructed by software, containing artefacts which themselves possess attributes*

### 2.1.2 Origins

The origins of virtual environments differ depending on how broadly the term is interpreted. The range of potential interpretations depends on many things, including a subjective interpretation of the required realism of the simulation. For example, it might be assumed that the simulation has to be realistic, which is not part of the definition in Section 2.1.1.3. It also depends on the particular definitions of the terms *software*, *region* and *artefact*.

It can be argued that Multi User Dungeon systems (MUDs)[29] were in fact basic virtual environments[38]. They enable a group of people to collaborate and communicate in a text based environment, whilst attempting to achieve goals and objectives<sup>4</sup>. These systems reached the height of their popularity in the early 80s, and contained many of the components found in modern virtual environments such as rooms and users<sup>5</sup>.

Currently, there exists a strong association between virtual environments and the concept of virtual reality (VR). VR represents technologies related to the input/output capabilities of a system. With simple MUD based systems, input and output was in the form of text. Input usually took the form of a domain specific language<sup>6</sup>, DSL, which closely related to natural language. This DSL would allow the user to describe his or her actions within the environment. Output was usually in the form of text descriptions of the rooms and objects and events that took place. VR attempts to bring the input and output mechanisms of the system closer to reality. Instead of text based inputs we might want to use our body movements and voice, and instead of text based outputs we might want to see, hear and feel the world. These are clearly very bold goals, and above and beyond the scope of this thesis, yet great strides have been taken since the early text based days. Some of the first virtual environments to include elements of virtual reality were war simulations such as the SIMNET system[22] which was designed and developed by Defence Advanced Research Projects Agency, DARPA.

<sup>3</sup>The word artefact is chosen instead of object to reduce the chance of confusion between a virtual environment object and a software object such as those found in object-oriented programming.

<sup>4</sup>Typically in fantasy setting populated with warriors, elves, dragons and pots of gold.

<sup>5</sup>The users can be controlled both by people and the system. A distinction discussed further in Section 3.1.3.

<sup>6</sup>For more information on DSLs see Section 2.3.

### 2.1.3 Range of Usage

#### 2.1.3.1 MUDS

Over the past 30 years, since the inception of the first MUD<sup>7</sup>, the usage of virtual environments has varied widely. MUDs were typically used for role-playing. The types of role-playing varied from Dungeons & Dragons[28] style fantasy worlds, to science-fiction visions. Players would communicate inside these worlds, interacting with each other and objects of various kinds. Often there would be objectives to achieve, but these were not necessarily essential to the experience. They are still in use, however, for collaborative work, communication[26] and even studies into the formation of online cultures[100].

#### 2.1.3.2 War Simulations

Simulating military scenarios was one of the first uses of virtual environments. During the 1980s the Defence Advanced Research Projects Agency, DARPA, developed SIMNET[22] which was designed to support up to several hundred simultaneous users. The users interacted with mocks of vehicles such as tanks and aircraft in order to simulate actual conditions. It was used for tactical rehearsals for military operations such as U.S. actions in Desert Storm in 1992.

#### 2.1.3.3 Gaming

An increasingly popular use of virtual environment systems is for entertainment. Gaming virtual environments tend to target consumer hardware such as PCs or consoles. Doom[53], released by id software in 1993, is considered to have pioneered the use of immersive 3D graphics and networked multiplayer gaming on the PC platform. Doom supported four simultaneous users allowing them to play either co-operatively or against each other. Since then there has been a consistent development of similar games offering increasingly realistic 3D graphics, and increasingly complex forms of collaboration with increasing numbers of participants (see Figure 1.1 in Section 1.1.1).

Today, all current games consoles and computers offer network support. There are a number of popular virtual environment systems offering a wide range of game experience, from World of Warcraft[57] to war simulations such as Return to Castle Wolfenstein[55] and to much more general community-driven simulations such as Second Life[58].

#### 2.1.3.4 Television and Entertainment

Television and entertainment is a promising direction for virtual environments. However, there hasn't yet been a media packaging of such technologies that has been a success in the same sense as popular

---

<sup>7</sup>The first known MUD was created in 1978 by Roy Trubshaw and Richard Bartle at Essex University in the UK on a DEC PDP-10.

television soaps or films. The following are examples of virtual environments used within television and entertainment:

- The MASSIVE-2 system[45] was used to research network patterns and user activity inside inhabited TV events for an experimental TV show called “Out of This World”.
- Counter Strike [116] is a game where teams of counter-terrorists battle against a team of terrorists in a series of rounds. Each round is won by either completing the mission objective or eliminating the opposing force. It is not only possible to play the game, it is also possible to be a spectator, watching the game from any viewing angle - including from the perspective of any of the actual game players. This is a popular option - particularly with high-profile battles.
- In 1993 Craig Charles hosted a television programme called Cyberzone which featured on BBC2. This game featured a number of contestants who battled each other in a virtual environment. However, it was not particularly popular, and only lasted for one series.
- The ITV television programme ‘The Krypton Factor’ put contestants through *“the ultimate mental and physical tests”*. From the 1988 series onwards, one of the tests, the response test, consisted purely of flight simulator tests. In these tests, the contestant interacted with a virtual environment simulating the landing, launching or flying of a variety of aircraft including helicopters, aeroplanes and even space rockets.

#### 2.1.3.5 Collaboration

Virtual environments can be used for cooperative or collaborative work between groups of people. A term often used for this is Computer Supported Cooperative Work (CSCW)[47]. For example, virtual environments have been used for team training exercises[77]. Modern commercial virtual environments such as World of Warcraft and Second Life have been used as research contexts for analysing collaborative play[88], and even economic studies[92].

There are a number of concurrency challenges to this field associated with object access issues such as locking, transaction mechanisms, turn-taking protocols, centralised controllers, dependency-detection, reversible execution, and master entities.[18]. It is issues like these which promote the thinking that virtual environments mimic many aspects of operating systems[74].

#### 2.1.3.6 Training

Virtual environments can be used for a variety of training purposes. The following are some examples of these:

- Virtual environment technology was used to construct a model of the Hubble Space Telescope. It was used to train a team for a repair and maintenance mission conducted by the National Aeronautics and Space Administration (NASA)[70].
- Virtual environments can be used to train new remote operation vehicle (ROV) pilots.[97] ROVs can be used for a variety of tasks that aren't necessarily accessible or safe for humans such as underwater search and salvage, inspection, surveying, scientific exploration, and disarming mines.
- Virtual environments can be used in the initial training of pilots in potentially dangerous or hazardous vehicles such as planes, tanks, and other military vehicles[81]<sup>8</sup>.

### 2.1.3.7 Industrial Design

Virtual environments are used in various fields of industrial design such as car manufacturing. For example, they can be used during the process of conceptualising and prototyping products[118]. In these cases the realism of the virtual environment interface can often be crucial[3]. Also, they may facilitate the collaborative design of new products[65].

### 2.1.3.8 Archaeology

VITA (Visual Interaction Tool for Archaeology), is an experimental collaborative mixed reality system for offsite visualisation of an archaeological dig[14]. VITA augments existing archaeological analysis methods with new ways to organise, visualise, and combine the standard 2D information available from an excavation (drawings, pictures, and notes) with textured, laser range-scanned 3D models of artefacts and the site itself. Virtual environments can also be used to recreate historical buildings or areas in order to help people visualise, and learn about past structures and contexts[40].

## 2.2 Interest Management

This section describes the concept of interest management in detail following its introduction in Section 1.3.2. Sections 2.2.1 and 2.2.2 introduce some of the key motivations and definitions of interest management found within the literature. Section 2.2.3 follows by describing a range of interest management methods used within virtual environment implementations. Finally, Section 2.2.4 shows how the techniques introduced in Section 2.2.3 can be distilled into a small number of core techniques. These core techniques will be the focus of the rest of this thesis.

---

<sup>8</sup>Flight simulators have even been used as a therapy technique for people suffering from a fear of flying[51].

## 2.2.1 Scalability of Virtual Environments

Section 1.2 described how in naïve virtual environments, the number of messages needed to be sent by the server increases quadratically as the number of users increases. Scalability can be an issue for large scale virtual environments. Brunton et al.[20] argue that very large scale distributed simulations suffer from two scalability issues with respect to network traffic:

1. sheer volume of data,
2. the ability to receive and process information.

The following are some of the methods used to tackle these issues:

**Load Sharing** Chen et al.[24], and Iimura et al.[56] split the computational load of the virtual environment system over a set of separate servers. Duong and Zhou[34] have developed load sharing algorithms that aim to optimise the spreading of computation load across a set of such separate servers.

**Caching and Pre-fetching** These techniques allow users to cache items locally (thus removing the need to request the same item again), and also to predict which items may be requested, and download them when it's optimal to do so (i.e. when the network is not busy)[94]. For example, the Cyberwalk system[89] supports caching and offers different levels of detail of geometry information through its multi-resolution caching system.

**Aggregation** In the PARADISE system[106] everything is an aggregate. An aggregation is a simulation entity that represents a group of other entities. By treating a number of entities as one artefact, the number of messages that needs to be sent reduces in proportion to the granularity of the aggregation.

**Peer to Peer Architecture** It is also possible to use peer-to-peer techniques, essentially breaking out of the traditional client-server architecture. This can have the effect of distributing and sharing the system's computation and messaging across multiple nodes, instead of relying on one single node. For example, Rhalibi et al.[101] propose a combination of a peer-to-peer architecture and caching and pre-fetching techniques.

**Multicast Communication** Many virtual environment implementations make use of the multicast protocol[35] as a way of reducing the number of messages that have to be sent. For example, Araujo et al.[5] propose a number of approaches to optimise the use of multicast groups within virtual environments by minimising the number of active multicast groups.

However, in terms of cost, network resources still remain very expensive compared with computational resources[48]. There is therefore an argument for the use of top-down approaches (see Section 1.3.2) to

efficiently manage network resources. One such top-down approach is interest management. Two of the main goals of interest management are to minimise network traffic and reduce the burden on clients[76] - the two main issues raised by Brunton et al. Interest Management is therefore a key technique to address the scalability issues of large scale virtual environments[69]. The following section will introduce and discuss the various definitions of this technique.

## 2.2.2 Definition

There are a number of terms that are often used interchangeably with the term interest management, such as *data distribution management*[109], *data subscription*[21] and *relevance filtering*[9]. The terms *data distribution management* and *data subscription* tend to represent interest management techniques focussed on the class of problems described in Section 2.2.1: namely scalability concerns. There is, however, another focus for interest management: namely the *management of interests*, i.e. the ability to represent and manipulate a variety of interests. The following sections will explore these different definitions.

### 2.2.2.1 Interest Management for Scalability

Ding and Zhu[33] describe interest management as *“the problem of avoiding broadcast communication”*. Brunton et al.[20] also provide a similar definition: *“where data is transmitted only if there is a defined need for it”*. These definitions clearly have a focus of reducing the number of data transmitted by the system - essentially a reduction in the number of messages sent. The correlation between the term *data distribution management* and the goal of scalability is made explicit when considering the goals of the HLA-DDM<sup>9</sup>[85]. This particular data distribution management system limits the messages received by users in order to reduce the message traffic over the network, and the data set required to be processed by the receiving user.

Brunton et al.[20] describe two primary purposes for the technique of interest management:

- **Technological:** to reduce the amount of network traffic
- **Operational:** to support the need for the user to define what information for the simulation will be displayed

The technological purpose matches the scalability motivations that were discussed above and also in Section 2.2.1. However, the operational purpose matches up with a new concept: that of user requirements. The following section will explore this particular concept in greater detail.

---

<sup>9</sup>HLA-DDM - High Level Architecture Data Distribution Management, introduced in Section 2.2.3.1



### 2.2.2.2 Interest Management for Managing Interests

Consider the following definitions of interest management:

- *“limiting the amount of information passed across a communications interface to the information of interest for a certain user perspective at that moment in time”*(Singhal and Zyda[105]),
- *“identifying which objects and information in a system are of relevance to a particular observer”*(Purbrick and Greenhalgh[99]),
- *“the process by which one exploits the interest of each user to minimise the number of update messages that must be propagated”*(Minson and Theodoropoulos[84]),
- *“the process of filtering irrelevant messages”*(Masa and Žára[76],
- *“reducing messages to a smaller relevant set”*(Morse[85]).

These definitions introduce a new concept - that of relevance or interest. This notion is succinctly represented with either Morse’s or Masa and Žára’s definition. This smaller, relevant set is often called an area of interest and usually correlates with the sensing capabilities of the system being modelled, such as visibility[20]. Meehan[83] describes the notion of an area of interest as *“the distributed parts of the virtual environment to which a user has access.”* These distributed parts may include artefact update data, artefact geometry and communication. This indicates that the definition of the concept interesting may not just be the representation of the interests of one particular user, but potentially the combination of many interests such as that of the user, and that of a system controlling access. This concept is described further in Section 3.4.

Antunes et al.[4] argue that the goal of interest management is to allow each user to only process the information that is relevant for them, and is used to:

- reduce network bandwidth,
- increase the system’s scalability,
- promote collaboration by using it to scope user interaction.

Therefore, by focussing on the management of interests, it is possible to tackle the issues of scalability described in Section 2.2.1, and also open up interesting areas of research such as techniques for describing interests, and the study of interaction scoping<sup>10</sup>.

---

<sup>10</sup>Techniques for scoping interaction are further discussed in Sections 2.2.3.3 and 2.2.3.5

### 2.2.2.3 New Definition

Consider Morse's definition of interest management:

*"reducing messages to a smaller relevant set"*Morse[85]

This statement, like most of the definitions discussed in Section 2.2.2 uses the message as the focus of interest. This may be due to the fact that interest management techniques were initially created to reduce the number of messages sent. However, consider the following definition:

*"identifying which objects and information in a system are of relevance to a particular observer"*(Purbrick and Greenhalgh[99])

Purbrick and Greenhalgh mention objects, or artefacts, as the focus of interest. As Section 3.1.2.2 introduces, events are changes in artefacts, and messages contain event information. If we use artefacts as a focus of interest, it is therefore possible to scope all the messages to those that affect the artefacts that we're interested in.

Section 2.2.2.1 described the relationship between terms such as *data distribution management* and the motivation of increasing scalability. Section 2.2.2.2 described the relationship between terms such as *interest management* and the motivation of managing interests. Perhaps another, potentially oversimplified<sup>11</sup> method of distinguishing these two concepts is that data distribution management is primarily concerned with messages, and that interest management is primarily concerned with artefacts.

Therefore, in order to disambiguate from terms such as data distribution management, my definition of interest management is as follows:

*"Interest management is the scoping of all world artefacts to a smaller relevant set"*

## 2.2.3 Techniques for Managing Interests

This section will introduce the range of techniques used for managing interests within virtual environments. These techniques are categorised in Section 2.2.4.

### 2.2.3.1 Class and Value Based Filtering

Class and value based filtering techniques define interesting objects based on logical predicates that reason about artefact attribute values, and artefact classes. For example, all red artefacts, and all vehicles are examples of predicates that reason about values and classes respectively.

The High Level Architecture - Data Distribution Management, HLA-DDM, uses this technique[108]. The HLA is a IEEE standard of computer simulation in 2000, developed by Defence Modelling and

<sup>11</sup>Clearly, there are exceptions to these rules. For example, it is perceivable that a user might wish to express an interest in a particular type of event. This is discussed further in Section 8.3.6.

Simulation Office (DMSO) of the U.S. Department of Defence. It was an initiative targeted at unifying almost all existing military simulations[85].

MASSIVE 2[43] supports the representation of a hierarchy of groups or classes. For example, a crowd may be composed of artefacts and other crowds recursively. This is similar to the concept of aggregations introduced in Section 2.2.1, however in this case the layers of abstraction, or aggregation, can be used when describing interests.

### 2.2.3.2 Domains

e-Agora[76] organises the shared state of a virtual environment into domains and sub domains. The domains represent categories of areas of interest (logical groups, regions, etc.), and the sub-domains represent concrete areas. Any state variable belongs to any number of domains specified upon creation (a static relationship). When a state variable is updated, a sub-domain set is associated with the update (a dynamic relationship). Users are then able to express an interest in a set of domains and sub-domains.

### 2.2.3.3 Interaction Analysis

Han et al.[48] introduce a filtering scheme that reduces the number of messages by dynamically grouping users based on their interests and relative distances. This approach attempts to make the following assumptions based on real world observations:

1. people can perceive artefacts near to them more frequently than those far from them
2. people focus more on objects of high interest
3. people tend to interact more frequently with people with similar interests

Ding and Zhu[33] analyse so-called “*crowd effects*”: how users interact in a crowded space. Their work is concerned with dynamic interaction in crowds, from which they derive the semantics of user behaviours, or more specifically, the alteration of interest focus of users. From examining the interaction, they are able to determine the following types of artefact:

- **Hotspot:** an artefact with many other artefacts interested in it,
- **Activist:** an artefact interested in many other artefacts.

They argue that these artefact types influence the psychology (and therefore interest) of the crowd through three main effects:

- **Propagation:** if  $A$  is interested in  $B$ , and  $B$  is interested in  $C$ , then  $A$  may become interested in  $C$ ,
- **Feedback:**  $A$ 's interest in  $B$  is affected by  $B$ 's interest in  $A$ ,

- **Conformity:**  $A$ 's interest in  $B$  is enhanced if  $B$  is a hotspot.

#### 2.2.3.4 Virtual Parallel Worlds

VELVET[32] introduces an adaptive mechanism which allows each participant to receive as much as possible (or requested) from the virtual environment. This is achieved by managing an area of interest through the concept of the parallel virtual world. Each user in VELVET has their own parallel virtual world, which is essentially a subset of the full world scoped by their interests. These interests are sets of targets for given metrics such as the number of users, the distance, the network bandwidth, or even a mixture of targets. Given that some of these metrics are constantly changing, such as the number of nearby users, the area of interest will also change accordingly. Oliveira[32] describes this as area of interest shrinking.

#### 2.2.3.5 Awareness

The following are examples of awareness used for representing interests:

- In the MASSIVE system[46] each virtual environment artefact has an associated aura which defines a spatial area within which interaction with other artefacts is possible. Interaction between two artefacts can therefore only occur if their auras collide or overlap.
- The HLA DDM system[85] employs a similar, but not necessarily spatial, concept through the notion of regions. Each artefact has both an update region and a subscribe region. An artefact is discovered by a user when the artefact's update region overlaps the user's subscription region.
- Benford and Fahlén's[12] introduced the concept of the spatial model of interaction. This model uses the following aura like entities:
  - **Focus:** an area representing an artefacts ability to perceive.
  - **Nimbus:** an area representing an artefacts ability to be perceived.

Each artefact in a virtual environment using this model has both a focus and a nimbus associated with it. These entities are used to determine awareness. Given two artefacts,  $A$  and  $B$ ,  $A$ 's awareness of  $B$  is proportional to how much  $A$ 's focus overlaps  $B$ 's nimbus.

- The e-Agora system[76] partially adopts the aura-nimbus model, but provides an abstraction which depends on relationships between properties that are not necessarily spatial properties. Auras and nimbuses refer to sets of domains and sub-domains (see Section 2.2.3.2).

### 2.2.3.6 Cells

A virtual environment may be divided up into tessellating regions often called cells. An example of this technique is found within the NPSNET system[73],[72]. NPSNET partitions the spatial area of the virtual environment into hexagonal cells. Each user has a circular area of interest, and if this area overlaps a cell, then the cell is deemed to be interesting. A user is therefore only interested in the cells within close proximity. Second-Life also splits the world into a set of tessellating squared regions[102].

### 2.2.3.7 Locales

Locales are spatial areas within a virtual environment, and can be used to define interest. For example a user may be interested in all artefacts within the same locale as itself. The following are examples of locales within virtual environments:

- In the Spline system[119], the world is divided into locales. Unlike the hexagonal cells in NPSNET, locales can have any shape, and they each have their own coordinate system<sup>12</sup>.
- The Score system[67] divides the world into cells. Similar to NPSNET, each user has an area of interest<sup>13</sup>, and if this area overlaps a cell, then the cell is interesting. Score differs from NPSNET in that it facilitates the dynamic re-partitioning the world<sup>14</sup>. Score allows for two policies which determine cell size: pre calculation of a fixed cell size, dynamic re-estimation of cell size during run time.
- Chen et al.[24] also dynamically re-partition the world into locales. The repartitioning is triggered by quality of services drops below an accepted level
- The MASSIVE-3 system[99] extends Spline's locales with support for abstractions. These abstractions may be sets of locales, or alternative representations of locales.

### 2.2.3.8 Visibility Based Filtering

The RING system[39] filters interest based on the visibility of a given artefact. This visibility is essentially a viewing frustum<sup>15</sup> with a given orientation. If another artefact is within the frustum, then that artefact is interesting. RING also pre-calculates line of sight visibility within the world based Teller and Sequin's[110] visibility pre-processing work. This allows occluding artefacts such as walls to interfere with the visibility of artefacts. Hosseini et al.[52] argue that it is possible to get the client's renderer to do

<sup>12</sup>Links between locales includes a 3D transformation for the coordinate systems. These transformations allow tardis like structures which may larger on the inside than the outside.

<sup>13</sup>In the case of Score, the area of interest is a square.

<sup>14</sup>Re-partitioning in this case is to tackle crowding issues (see Section 2.2.4.2 for more information on crowding).

<sup>15</sup>A viewing frustum is a representation of the visibility of an artefact. This representation is typically a cone or pyramid with a particular size and orientation.

this work in order to remove load from the server. However, this has no reduction affect on the number of messages sent within the system.

## 2.2.4 Interest Management Techniques Categorised

This section describes a mapping between the interest management techniques introduced in Section 2.2.3 and the following generic categories:

- Categorisation (see Section 2.2.4.1),
- Locales (see Section 2.2.4.2),
- Interacting Locales (see Section 2.2.4.3),
- Hybrid Approaches (see Section 2.2.4.4)

Finally, section 2.2.4.5 describes the mapping itself.

### 2.2.4.1 Categorisation

The category based approach is a more general form of the class and value techniques described in Section 2.2.3.1. Category based filtering techniques determine interesting artefacts based on logical predicates that reason about information associated with each artefact. The following are the different types of information that may be associated with an artefact:

**Artefact Attribute Values** This information is essentially the data that the virtual environment needs to represent and use the artefact. For example, it might be geometry information, location and orientation information etc.

**Explicit Metadata** This information is extra information about a given artefact that is defined like attribute values, yet it is only used for defining interest statements. This information could be class associations, hierarchy information etc.

**Implicit Metadata** This information is similar to explicit metadata. However it is not defined like attribute values, it is generated using algorithms based on other information. This information could be levels of interaction, or a description of the artefacts within line of sight.

### 2.2.4.2 Locales

As Section 2.2.3.7 describes, locales are spatial areas within a virtual environment. Locales can therefore be used to describe regions such as areas, regions and cells. Purbrick and Greenhalgh[99] also argue that region-like notations such as cells and viewing frustums can be subsumed by a mapping on to locales.

For example, it could be possible to use a spatial area such as a frustum to model the area of visibility of an entity within the virtual environment. This frustum can be seen to be a particular type of locale.<sup>16</sup>

### 2.2.4.3 Interacting Locales

Given that locales can represent spatial areas, it is therefore possible to use them to represent auras, focuses and nimbuses. We are therefore interested in the spatial relationships or interactions of these particular types of locales. For example, if the locale that represents artefact *A*'s focus overlaps the locale that represents artefact *B*'s nimbus, then artefact *A* can be said to be aware of artefact *B*.

### 2.2.4.4 Hybrid Approaches

Each of the three techniques introduced above has its own limitations. For example, consider the following issues:

**Categorisation** It can be hard to classify users by predefined types or classes. Han et al.[48] argue that this kind of filtering does not work well with systems where users' interests change dynamically.

**Locales** Ding and Zhu[33] argue that region based techniques do not handle crowded situations. For example, they tend to assume an even distribution of artefacts, yet in certain contexts the artefacts might all crowd in a particular locale<sup>17</sup>.

**Interacting Locales** May suffer from similar issues to locales such as crowding.

We may therefore wish to combat some of the limitations that are faced by individual techniques by combining the techniques together. For example, it might be possible to combat the effects of crowding by combining locale based interest with a category based interest.

### 2.2.4.5 Technique Mapping

Table 2.1 describes how the various techniques introduced in Section 2.2.3 may be mapped on to the categories introduced in Section 2.2.4. This thesis is based on these three types of interest management technique: categorisation, locales, interacting locales, and a mixture of all three techniques.

These general techniques will be the subject of the rest of this thesis. Any formalisation or implementation of these general techniques can therefore be used to represent any of the more specific techniques described in this chapter. The formalisation of these general techniques, lends itself to the specification of a new domain specific language which can focus entirely on representing interests. Domain specific languages are discussed further in Section 2.3.

<sup>16</sup>It is important to note that locales in this sense differ from those presented in Spline and MASSIVE 3 in that they have a global co-ordinate system, and that presence within a locale is a fundamental property.

<sup>17</sup>For more information on crowding within virtual environments see [87], [86] and [13].

Table 2.1: Categorisation of Interest Management Techniques

Technique	Category	Method
Class Based Filtering	Categorisation	Filter by explicit artefact metadata representing its class
Value Based Filtering	Categorisation	Filter by artefact attribute values
Group Hierarchies	Categorisation	Filter by explicit artefact metadata representing its relationship to other artefacts
Domains	Categorisation	Filter by explicit artefact metadata representing its membership of the user's domains and sub-domains
Interaction Analysis	Categorisation	Filter by implicit artefact metadata based on statistics gathered from previous interactions
Virtual Parallel Worlds	Hybrid Approach	Filter by a logical combination of implicit artefact metadata, explicit artefact metadata, artefact attribute values and relationship with locales associated with the user (e.g. a viewing frustum).
Spatial Model of Interaction	Interacting Locales	Use locales to represent artefact auras. When the two locales representing auras collide, then the corresponding artefacts are interested in each other.
HLA DDM Regions	Hybrid Approach	Create two sets representing artefact update and subscribe regions by a logical combination of implicit artefact metadata, explicit artefact metadata, artefact attribute values and relationship with locales associated with the user (e.g. a viewing frustum). Check for artefacts that are members of both sets.
Cells	Locales	Create a set of tessellating locales representing the cells.
Locales	Locales	A direct mapping.
Visibility Based Filtering	Locales	Use locales to represent artefact viewing frustums.
Line of Sight	Categorisation	Determine which artefacts are visible using algorithms based on artefact attributes (geometry information).



## 2.3 Domain Specific Languages

Domain specific languages (DSLs)<sup>18</sup> are computer languages targeted to a particular kind of problem, rather than a general purpose language aimed at any kind of software problem.[37]. For example, DSLs are not necessarily Turing complete, i.e. possessing the computational power equivalent to the universal Turing machine[115]. The expressiveness or computational power of a DSL need not be any greater than the demands of context or domain within which it will be used. Some well-known examples of DSLs are HTML and SQL and XML.

**Internal DSLs** use the constructs and syntax of a general purpose programming language itself to define a DSL<sup>19</sup>. Constructing DSLs like this is similar to bottom-up programming[42] in which the language is changed to suit the problem. The Lisp, Smalltalk and Ruby communities have a strong tradition of using internal DSLs.

**External DSLs** have their own syntax. A compiler is then written to parse the DSL and possibly generate code for standard general purpose language, or interpret it directly. This approach is traditional in the Unix community which has many tools that make this easier such as yacc and lex[62]. XML is an example of an external DSL.

The expressiveness of a language refers essentially to the power and ability of representing different concepts within its domain or context. As explained above, the expressiveness or computational power of a DSL need not be any greater than the demands of context or domain within which it will be used. The focus of DSLs tends to be on creating a language that fits the domain for which it was created. The judgement of how well the language fits is clearly subjective. However, it usually has a correlation to the readability and succinctness of the language from the perspective of the domain within which it is executing. Sections 2.3.1 and 2.3.2 explore the concepts of readability and succinctness respectively.

### 2.3.1 Readability

In a given language, complex, expressive statements can very easily become difficult to read. A good example of this is the syntax of regular expressions. For example, consider the following regular expression which matches valid e-mail addresses:

```
/^A([w\.\-\+])@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/i
```

Not only are more readable statements easier to read<sup>20</sup>, they are also easier to write and verify. Having a readable language can also help in exposing obvious bugs. For an example of a language that

<sup>18</sup>DSLs are also called little languages or mini languages[64].

<sup>19</sup>Internal DSLs are also referred to as embedded DSLs.

<sup>20</sup>Please excuse the illustrative tautology.

attempts to be more readable, consider COBOL. COBOL attempts to bring programming languages closer to natural languages using keywords such as MULTIPLY, GIVING and BY. Consider the solution

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad (2.1)$$

of the quadratic equation  $ax^2 + bx + c = 0$  written in COBOL<sup>21</sup> looks as follows:

```
MULTIPLY B BY B GIVING B-SQUARED.
MULTIPLY 4 BY A GIVING FOUR-A.
MULTIPLY FOUR-A BY C GIVING FOUR-A-C.
SUBTRACT FOUR-A-C FROM B-SQUARED GIVING RESULT-1.
COMPUTE RESULT-2 = RESULT-1 ** .5.
SUBTRACT B FROM RESULT-2 GIVING NUMERATOR.
MULTIPLY 2 BY A GIVING DENOMINATOR.
DIVIDE NUMERATOR BY DENOMINATOR GIVING X.
```

The COBOL approach is probably more readable to someone without a background in mathematics. However, the mathematic equation is more succinct. Clearly, there is a balance to be struck between readability and succinctness which many believe COBOL never managed to successfully achieve. This is evident in the presence of humorous attempts to explain the expansion of the acronym COBOL to phrases such as *"Compiles Only Because Of Luck"*.<sup>22</sup> The concept and importance of succinctness is explored in the next section.

### 2.3.2 Succinctness

In addition to being readable, the language should also be succinct. One of the disadvantages of overly readable languages is that they tend towards natural language which can often be vague and open to interpretation. Consider the language AppleScript, which, in a similar fashion to COBOL, attempts to be readable by emulating natural language. For example, here is an AppleScript snippet which fetches the first paragraph of a document called 'Read Me' currently open in a text editing application called TextEdit<sup>23</sup>

```
tell application "TextEdit"
    get paragraph 1 of document "Read Me"
end tell
```

<sup>21</sup>ignoring (for didactic purposes) the existence of the 'compute' verb which allows: COMPUTE X = (-B + (B \*\* 2 - (4 \* A \* C)) \*\* .5) / (2 \* A)

<sup>22</sup> "Each language has its purpose, however humble. Each language expresses the yin and yang of software. Each language has its place within the Tao. But do not program in Cobol if you can avoid it." [60]

<sup>23</sup>If in fact such a document exists and TextEdit is currently running with it open

There are three metaphors represented here: ‘talking’ to an application, ‘describing’ a set of information and ‘getting’ that information. However, in this particular context we only really need to scope and get the information. The following example written in rb-appscript, a Ruby replacement of AppleScript does exactly that:

```
app('TextEdit').documents['Read Me'].paragraphs[1].get
```

If read in reverse, it is clear that we’re getting the first paragraph from the document called ‘Read Me’ from the application called ‘TextEdit’.

A succinct language tends to be conceptually simpler, and typically easier to understand. However, to be understood at all it needs to also be readable. If the language is too conceptually simple, then it may not be expressive enough. Finally, if the language has no ability to abstract complexity, then complex statements will tend to be proportionally as complex as the concept they’re trying to represent - limiting the language to the complexity that the programmer is capable of manipulating in his or her mind at any one time, or being generated by a tool (which essentially manages the complexity).

## 2.4 Ruby

Ruby[27] is an open source object-oriented programming language. It was created by Yukihiro Matsumoto, まつもとゆきひろ, who designed it to not only make programming easy, but also fun<sup>24</sup>[112]. It is a fully object-oriented language, i.e. everything referenced by a variable is an object. This includes numerical values, boolean values, and even true, false and nil. This is unlike languages such as Java which offer a hybrid approach consisting of both objects and primitive types<sup>25</sup>.

Ruby inherits features from other languages such as: Lisp, Smalltalk, Perl and CLU. Such features include blocks (otherwise known as closures), singleton classes, excellent meta-programming[96] support<sup>26</sup>, and strict but dynamic typing<sup>27</sup>. Ruby supports a programming style commonly referred to as ‘Duck Typing’<sup>28</sup>. This is because the interpreter is generally only concerned whether an object responds to a particular method, rather than what type it is<sup>29</sup>. This can greatly simplify the handling of similar data structures, or types of information.

---

<sup>24</sup>Ruby also has fun documentation too [111].

<sup>25</sup>Consider the differences between the Java primitive `int` and the class `Integer`

<sup>26</sup>In concept, a program implemented with meta-programming methods is similar to a metacircular evaluator, such as the one found in Scheme[2]. With a metacircular evaluator the language is implemented with the same language, and with meta-programming the program is implemented (in part, or whole) by the same program.

<sup>27</sup>All Ruby objects have a type, but the language does not require you to specify types in method signatures or variable declarations

<sup>28</sup>If it walks like a duck, and quacks like a duck, then it’s a duck.

<sup>29</sup>This is similar to Gibson’s Affordance Theory[41], where affordance theory states that the world is perceived not only in terms of object shapes and spatial relationships but also in terms of object possibilities for action (affordances).

### 2.4.1 ERB

ERB is a lightweight templating system, allowing you to intermix Ruby code and plain text.[104]. It breaks its input text into checks of regular text and program fragments. Then it builds a Ruby program that, when run, outputs the result text and executes the program fragments. Program fragments are enclosed between `<%=` and `%>` markers.

## 2.5 Ruby on Rails

Ruby on Rails[91][113] is a full-stack open source web framework initially written by David Heinemeier Hansson , and now maintained by a core team of developers. It provides scripts that set up a skeleton framework that provide a working foundation for a project. Two core parts of the Rails framework were used extensively for the implementation of the ideas presented within this thesis. These were the database object relational mapper Active Record[49] and the Ruby extension library Active Support[50].

### 2.5.1 YAML

YAML<sup>30</sup> is a straightforward machine parsable data serialisation format designed for human readability and interaction with scripting languages such as Perl and Python and Ruby[59]. It allows the representation of the basic data types common to most high-level scripting language such as lists, hashes and scalars. It uses significant whitespace to denote hierarchy and structure<sup>31</sup>.

### 2.5.2 RSpec

RSpec[7] is a behaviour driven development framework<sup>32</sup> for the Ruby programming language. It provides programmers with a DSL to describe the behaviour of Ruby code with readable, executable examples that guide you in the design process and serve well as both documentation and tests. For an illustration of how RSpec can be used to describe the behaviour of a system see Appendix A.

## 2.6 Limitations of current practice

This section aims to introduce and describe the limitations in the current virtual environments. This will be achieved by arguing that the main focusses and goals of current virtual environments make assumptions that do not hold for all simulation types, and that there are in fact other goals that users and systems may need to focus on, such as adaptability.

<sup>30</sup>YAML is a recursive acronym for YAML Ain't Markup Language, and when spoken rhymes with camel.

<sup>31</sup>Ken Arnold argues the importance of significant whitespace. He believes that fixing the usage style of a language in syntax is a good thing[6].

<sup>32</sup>Behaviour driven development is discussed further in Section 4.3.2.2.

## 2.6.1 Assumptions Made

As introduced in Section 1.1.1, one of the primary challenges for virtual environments is scalability. Virtual environment designers have aimed to optimise their designs in order to increase the possible number of participants and interactivity between those participants. Their visions have taken a wide variety of forms as summarised in Section 2.1.3. These are laudable goals, and indeed there is much commercial benefit from massively scaleable virtual environments, particularly in the Massive Multiplayer Online Gaming (MMOG) sector. In achieving these goals however, the designers have had to make many assumptions about how the system is used and works (as introduced in Section 1.1.2). These assumptions are a fundamental part of the optimisation process, and by making their systems more optimised the designers aimed to produce an increase in scalability. The major types of assumption can be represented with three categories: assumptions of interest, assumptions of capabilities and implicit assumptions.

### 2.6.1.1 Assumptions of Interest

The various interest management techniques introduced and categorised in Section 2.2 are all examples of assumptions of interest made in the design stage of a virtual environment. As the intended usage of the virtual environment is decided in the design stage, many assumptions on the interests of the user and system can be fairly made. For example, in a war simulation it would be fair to say that the users are going to be interested in things that can kill them. It may also be fair to say that as war simulations attempt to simulate reality the interests should be limited to what it would be realistic to be interested in. This would mean that a soldier on the ground would not be allowed to be interested in anything that it couldn't see or hear.

### 2.6.1.2 Assumptions of Capabilities

When most virtual environments are designed they are targeted at a particular system or set of systems. For example an online multiplayer computer game will ship with a description of the minimum system requirements necessary to be able to use the product, other virtual environment implementations such as certain bespoke war simulations may be even more restrictive. These restrictions or assumptions of capabilities allow the virtual environment designer to make yet more optimisations when implementing the system. Examples of the kinds of assumptions a designer may make are the following:

- a minimum network bandwidth
- processor speed
- available RAM
- input capabilities

- output capabilities
- the availability of audio, graphic or physics accelerator cards

For example, the Macintosh version of World of Warcraft has the following minimum set of specifications:

- OS X 10.3.5
- 933 MHz or higher G4 or G5 processor
- 512 MB RAM or higher
- DDR RAM recommended
- ATI or NVIDIA video hardware with 32 MB VRAM or more
- 4 GB or more of available hard drive space
- 56k or higher modem with an Internet connection

### 2.6.1.3 Implicit Assumptions

As introduced in Sections 2.6.1.1 and 2.6.1.2, assumptions are made both over the user's interest and the system capabilities. These are clear and obvious design decisions, and the fact that there are a variety of interest types (see Section 2.2.4) and target systems indicates the possibility that there may not be one general solution. This issue is further backed when we consider that all of the design assumptions made are all in terms of some implicit assumptions. These implicit assumptions are hidden in the simulation definitions and generally fit under the banner that the user wants optimal performance and increased realism. Examples of implicit assumptions are as follows:

**More Frequent World Updates** If the world is extremely dynamic with many events per unit of time, then as many of the updates as possible should be sent to the client. The aim being that the client's view be as dynamic as the representation of the world in the server.

**More Detailed Worlds** Where the world offers a choice in detail or Level of Detail (LOD) the client wants the highest available detail for each artefact.

**More Artefacts Visible at any one Moment** The maximum number of artefacts capable of being displayed should be as high as possible. With a greater number of artefacts visible, the granularity of the system can be reduced and therefore provide a potential increase of realism (given the assumption that the simulation is attempting to simulate a world with a large number of small particles such as ours, rather than a world with a small number of large particles).

**Greater Responsiveness** The amount of time taken for users to be aware of events needs to be as small as possible.

**No Network Bandwidth Wasted** The system should make full use of the network bandwidth, on the assumption that idle or underused bandwidth means that the client's view is not as frequently updated, detailed, rich or responsive as it could be. These are again implicit assumptions of what the client wants.

## 2.6.2 Problems with Assumptions

Assumptions are only a good thing as long as they continue to hold true. If an assumption is no longer valid then it can actually reduced the effectiveness of the system if no assumptions had been made. To help understand this concept it is useful to take a quick look the association between assumptions and interests, and then a discussion of why interests may change.

### 2.6.2.1 Assumptions and Interests

So far, the distinction between assumptions and interests hasn't been very clear. The concept of Interest Management was defined in Section 2.2.2, and then in Section 2.2.1 the motivations were discussed. Interest Management as defined then has many similarities with assumptions of interest (as introduced in Section 2.6.1.1). So what of the other assumptions:

- assumptions of capabilities (Section 2.6.1.2)
- implicit assumptions (Section 2.6.1.3)

How do these assumptions relate to our definition and motivations of Interest Management? The answer to this question is rather subtle and requires a closer and more detailed look at the earlier definitions of Interest Management and assumptions. When Interest Management was introduced in Section 2.2.2, it was defined as an algorithm or set of algorithms that reduce the set of all world artefacts and events to a smaller more interesting set, the motivation being that if all objects and events were visible to the client then there would be a serious impact on the scalability of the system. Assumptions were introduced in Section 2.6.1 as design decisions made in the early stages of a virtual environment. An initial interpretation of these definitions may indicate that interests are driven by the user, and assumptions are driven by the system. For example, MASSIVE 1's interest options are actively and explicitly manipulated by the user of the system. However, when these two concepts are looked at in terms of scalability their definitions seem to be reversed, for the assumptions are mostly the goals of the user (the implicit assumptions) limited by the capabilities of the user (assumptions of capabilities), and the interests are the optimisation techniques used to deliver those goals. In terms of implementation, assumptions and

interests are identical in all of the current virtual environments. They are either implemented as hard-coded rules, or hardware. Essentially they are both static. If, however, we view the concept of interest as user-driven it might therefore be dangerous to assume that a user's interest would never change. Similarly, it may be very restrictive to assume that the system capabilities never change.

### 2.6.2.2 Changes in Interest

As suggested at the end of Section 2.6.2.1, when we consider that interests could be created by the user it would be dangerous to assume that those interests will not change. In addition, we might want to consider the possibilities that present themselves if interests were allowed to change, and what situations may require a change in interests. Examples of situations that may require a change in interests are:

**Rules Change** It might be strange to consider virtual environments as having rules, but whether explicit or implicit, they do exist. Explicit rules are found in the form of game rules, such as those found in online multiplayer virtual environments. Implicit rules are those that are required for the virtual environment to simulate reality. Examples of implicit rules are: artefacts in a viewing frustum must be visible (simulating sight), items behind opaque artefacts must not be visible (simulating line of sight), artefacts not resting on another artefact must fall until it reaches an artefact below it (simulating gravity). Not all virtual environments have a set of rules that are static for their entire duration. Some virtual environments have a number of phases that may have an entirely different set of rules. It may therefore be impossible to create a generic set of interests that cover all of the rules of all of the phases at once. There may also be virtual environments where the rules are generated dynamically as the virtual environment progresses, and therefore there is no way that all of the rules of the game can be determined and defined at the start.

**Tactics Change** There are many reasons why the tactics used in a virtual environment may change, this may be due to a change of rules, due to a new understanding of the opponent, etc.

**Players Change** The number of players of a particular virtual environment may be dynamic; people may enter and leave a game. The number of players, or the existence of a particular player or set of players may have an effect on interest.

**Resources Change** The interests may want to be coupled with the capabilities or resources of a system. If the resource capabilities aren't very high, then the tactics that might be employed to deal with this may vary. For example, if a user has a poor graphics card and small network bandwidth, then they may either be sent low resolution models, have fewer models visible at once, or a mixture of both these options.

**Heterogeneous Interests** Not all users of the system may have the same interests. Maybe not all users are subjected to the same set of rules, and maybe they do not all want to employ the same



tactics. Not all users may have the same resource capabilities, therefore causing these interests to be different.

### 2.6.3 Proposed Solution to Interest Management

Static interests or assumptions (however they may be defined) are not always a good thing. Minson and Theodoropoulos[84] also argue that it is also possible that interests could be defined automatically at run-time. Macedonia and Zyda[74] define views of the virtual environment as either synchronous (where everyone sees the same) or asynchronous (where users have individual control over when and what they can see). Antunes et al.[4] argue that there are situations where different interest management policies could be more useful than one static policy. They also argue that an interest management solution must support the following abilities:

- the ability to define interests,
- the ability to support different interest management policies.

Section 2.6.2.2 has shown us that interests may change, and assumptions made early on may no longer be valid. In order to tackle these issues we need a system that can cope with changing interests and goals, and one where fewer assumptions have to be made. This would therefore create a more adaptable system.

The solution is a dynamic interest management technique, which is not only able to cater for user's interests, but also those of the system. This technique needs to be able to represent static interest management techniques as introduced in Section 2.2.3, and allow the interests to change during the lifetime of the system: at runtime if possible. This technique needs to also be capable of reasoning about resources and to cater for interests of adaptability.

However, before we can achieve this, we need to establish these goals and possible solutions in the context of the term virtual environment. Section 2.1.1.3 introduced the following definition:

*'A region, constructed by software, containing artefacts, which themselves possess attributes'*

This definition, although sound, is extremely generic. It would be very difficult to reason about interest management within the scope of this definition: doing so would require the definition of a lot of concepts that would commonly be associated with virtual environments such as events, and time. Section 3.1 develops this definition of virtual environments further, and Section 3.2 shows us how we can define interest statements to support the management of dynamic interests.

## Chapter 3

# A Framework for Dynamic Interest Management

As discussed in Chapter 2, interest management is used in different ways within virtual environments. The main purpose, however, has been to improve scalability. Section 2.6 discussed the limitations of static interest management techniques, and Section 2.6.3 proposed dynamic interest management as a solution to some of those limitations. This chapter will expand on the concept of dynamic interest management.

Section 3.1 will expand upon the definition of interest management proposed in Section 2.1.1.3, and introduce a conceptual model for virtual environments. This conceptual model will serve as the foundation for further discussion of interest, interest management, and, of course, dynamic interest management. Section 3.2 discusses the definition of statements which represent interest, Section 3.3 introduces some example interest statements, and finally 3.4 introduces issues raised when interests need to be constrained, and when they conflict with other interests.

### 3.1 A Conceptual Model for Virtual Environments

Virtual environments themselves can be seen as models in their own right. They contain a set of variables, or artefacts, and describe the set of quantitative and logical relationships between them. For example, consider a model of warfare such as NPSNET[73]. Such a model would describe a number of typical artefacts involved in warfare: buildings, tanks, planes, artillery, armaments, etc. It would also describe a number of possible interactions between these artefacts: a tank may destroy a building, a plane may bomb a tanks, etc. Through this model, the users would be able to reason about general concepts pertaining to warfare such as skills, tactics, and teamwork. This Section, however, is not concerned with the notion of virtual environments being models, but is concerned with defining a more general model of virtual environments.

Although a number of taxonomies of virtual environments already exist[74][23], it is hard to find a pure definition, or conceptual model of a virtual environment. This section aims to create an abstract definition, or model of a virtual environment for the purpose of reasoning about interest and dynamic

interests. Using this model, it will be possible to conceptualise interest within the context of a virtual environment, and then evaluate the feasibility of representing different types of interest within the model.

Section 3.1.1 will explore the motivations of creating a conceptual model of a virtual environment, and Section 3.1.2 will introduce the axioms, or underlying propositions, on top of which the conceptual model can be built.

### 3.1.1 Motivations

A model is a theoretical construct that represents a particular viewpoint or perspective of a concept. It is created with the purpose of reasoning about a particular set of qualities and relationships pertaining to that concept. When designing a model it is important to know the motivations of the model, and therefore which qualities or relationships you wish to reason about.

The model presented in this chapter has three main motivations: to reason about interest, to be a model of other virtual environments, and to be implementable. These motivations will be explored in the following Sections (3.1.1.1, 3.1.1.2, and 3.1.1.3).

#### 3.1.1.1 To Reason About Interest

The primary purpose of this model is to be able to reason about interests. It is therefore necessary for the model to represent *what* we want to be interested in, and *how* we define whether we're interested.

**3.1.1.1.1 What: Artefacts** It is important that the model defines the units that can be reasoned about in terms of interest. These units shall be described as the artefacts within the virtual environment. In this context, the term artefact has the following definition:

**artefact, *n.* and *a.***

**A. *n.*** Anything made by human art and workmanship; an artificial product. [61]

In using the term artefact, we will also draw a distinction from the term object found in object-oriented programming. Artefacts are described in greater detail in Section 3.1.2.2.

**3.1.1.1.2 How: Interest Statements** Given the definition of artefacts within our virtual environment, we need to have the ability to create statements which describe which of the available artefacts we are interested in. In order to achieve this we need to have a method or set of methods with which we can distinguish artefacts from other artefacts. Section 3.1.2.2 describes how artefacts consist of one or more attributes which have values which can be compared with the values of other artefact attributes. Using the presence, absence or value of artefact attributes we can describe the difference between artefacts. This affords us with the ability to describe the difference between the artefacts we are interested in,

and the artefacts we're not interested in. These descriptions are interest statements and are described in greater detail in Section 3.2.

### 3.1.1.2 To be a Model of other Virtual Environments

The model does not just need to reason about interest in general (as described in Section 3.1.1.1), it needs to reason about interest in the context of virtual environments (as defined by the range introduced in Section 2.1.3). The model therefore needs to be a model of virtual environments in addition to being a model of interest.

Virtual environments are more than a collection of artefacts with attributes. For example, virtual environments may change over time: artefacts may enter or leave, the values of artefact attributes may change. Also, information needs to flow between users and the system: users may interact with or directly influence artefacts, and may want to experience various aspects of the environment. The model therefore also requires the ability to reflect upon characteristics such as these.

### 3.1.1.3 To be Verifiable

In order to evaluate the claims this thesis makes with respect to dynamic interest management, the model needs to be verifiable. Verification in this case will be achieved by two methods. Firstly the model will be reasoned about, using the introduced set notation. Secondly, the model will be implemented (as discussed in Chapter 4). Using this implementation, data will be generated and evaluated by applying the implementation to a series of case studies.

## 3.1.2 Axioms

This section aims to describe the underlying propositions on top of which the conceptual model of a virtual environment can be built. Section 2.1.1.3 introduces the following definition for virtual environments:

*'A region, constructed by software, containing artefacts, which themselves possess attributes'*

From this definition we can determine that a virtual environment contains attributes and artefacts. Given that one of the motivations of interest management is to communication, we can also infer that virtual environments also contain events (the messages to communicate), processes (chains of events) and time (a temporal separator for the events). Therefore, at its most basic level an implementation of a typical virtual environment consists of attributes, artefacts, time, events and processes.

### 3.1.2.1 Attributes

An attribute is a named value. Each attribute belongs to a particular *data type*, and each *data type* has a set of possible *values*. For example, an attribute may be called *a*, the type may be called integer, and

the set of all possible values is the set of all integers ( $\mathbb{Z}$ ).

There should be a set of relationships between values of the same type such that two values can be compared. For example, with integers, the relationships are  $<$ ,  $>$  and  $=$ .

### 3.1.2.2 Artefacts

Also commonly called objects, artefacts are the individual units of the virtual environment; a virtual environment consists of one or more artefacts, and each artefact consists of one or more attributes[114]. For example a simple artefact may have the following attributes:

- A unique identifier,
- colour,
- x co-ordinate,
- y co-ordinate,
- z co-ordinate,
- geometric description.

The state of an artefact is the value of its attributes at a particular point in time in the system.

### 3.1.2.3 Events

An event is a change in the state of the system. This is essentially an alteration of one or more attributes of one or more artefacts at a particular time (discussed in Section 3.1.2.4). These alterations must appear to occur simultaneously, i.e. occur in the same logical time unit. Some examples of events are:

**A button is pressed** this may alter the colour attribute of the artefact.

**An artefact moves** this event may alter the artefact's x, y and z co-ordinate attributes.

Events may also describe the addition or removal of information within a virtual environment. This includes adding or removing attributes from artefacts, and adding or removing artefacts from the virtual environment. In a similar approach to that proposed by Hosseini et al.[52], it is assumed that these events are transmitted via a different channel to any other kind of message (i.e. visual data, voice communication etc.)

### 3.1.2.4 Time

In this model, time can be seen as a relationship between two events which orders them chronologically. On a simplistic level, this can be represented as a global number, and each event being associated

with a number. The ordering is then reduced to a numerical ordering relationship between the numbers associated with each event, or that of the global number.

### 3.1.2.5 Processes

Processes trigger events. The life of a process may span multiple time units unlike an event which is only active between time units, i.e. they happen instantaneously or in zero time. A process may be seen as a logical grouping of events that are inter-linked through a particular relationship. Some examples of processes are:

**A lift is called and moves between floors** this process may have been triggered by an event such as the lift button being pressed. It might create subsequent events that cause the lift to move to the appropriate floor, as well as the appropriate lights turning on at the appropriate times.

**A person stands up** this process may cause the artefacts that construct the person (the feet, legs, torso, arms etc) to initiate a movement, composed of multiple events, that span a number of time units resulting in an animation that imitates a real person standing up.

### 3.1.3 Users

So far, our conceptual model has consisted of abstract concepts which are internal to the virtual environment. In order for the virtual environment to have any relevance or use to our own external environment, then there must exist the ability for entities from an external environment to interact with the model. Section 3.1.4 will discuss the notion of interaction, and this section will focus on these external entities.

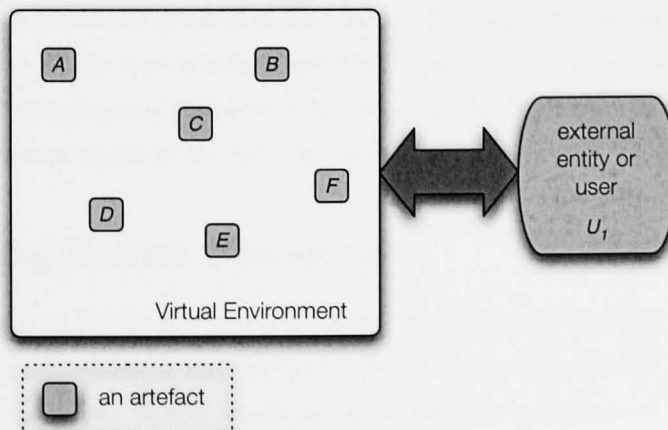


Figure 3.1: An External Entity or User

Entities external to the environment are called users. Typically a user is associated with an artefact, and such an artefact which has a direct association with a user is commonly called an avatar. For example, Oliveira[31] and Fuhrer et al.[38] define an avatar as a representation of a user in a virtual world. However, users are not always associated with artefacts, especially if the user is an anonymous observer or spectator in the virtual environment. There is typically a distinction made between dynamic entities (avatars) and static entities (artefacts)[5].

Figure 3.1 shows the relationship between the virtual environment and a user. As can be seen, the user ( $U_1$ ) exists externally from the virtual environment. One of the artefacts within the virtual environment may be associated with the user. For example, the user ( $U_1$ ) may be controlling the artefact  $D$ , in which case we can say that  $D$  is  $U_1$ 's avatar.

### 3.1.4 Interaction

In Section 3.1.3, we suggested that a user may be associated with an artefact. It is interesting to consider how that association occurs.

In the context of this conceptual model, when considering interaction, we only need to consider events. This is because time is not interactive (i.e. you cannot change its course), artefacts are just pure state, and therefore provide no means for interaction, and processes themselves are simply generated sets of events which can only be created by events.

Events in the real world may be directly mapped onto events in the virtual environment. Examples of these real world events acting as triggers are keyboard strokes, joystick movements, or motion detector sensors attached to real people. The ratio of these mapped events to processes describes how simulated the virtual environment is. For example, if each event in the virtual environment was mapped to a real world event, and there were no processes in the virtual environment, then this can be said to be very realistic. On the other hand, if we have only one event - a *start* event, and the rest of the events in the system were created by processes, then this can be said to be entirely simulated. The level of simulation of a virtual environment should be decided in the design phase, and is tightly coupled to its usage.

## 3.2 Defining Interest Statements

As introduced in Section 3.1.2.2, artefacts are the things that virtually exist within a virtual environment. Artefacts are the things we can be interested in: the cars, the people, the lifts, the buildings, the items that populate virtual worlds. Interest statements should reason about artefacts: we want to be able to describe the set of all artefacts that are interesting to us.

Our goal is to create a mechanism that allows us to reduce the set of all world artefacts to some definable subset (as shown in Figure 3.2). The statement which defines this subset is called an interest

statement. In order to create such a statement, it is necessary to have the language to describe it. Set theory is the perfect candidate for such a language. Set theory is one of the true foundations of mathematics, and can be used to formalise all mathematical concepts. It is sufficiently universal to formalise the notion of interest for our conceptual model. Section 3.2.1 introduces such a formalisation.

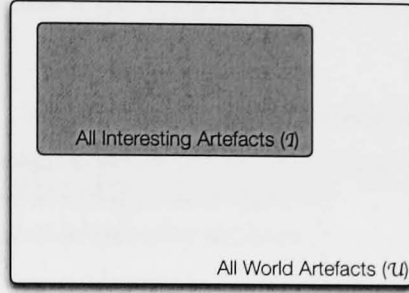


Figure 3.2: Interesting Artefacts

### 3.2.1 An Intensional Definition of Interest

One of the requirements of an interest management system is the ability to describe subsets of the set of all world artefacts. In order to understand this concept in greater detail, consider the following assumptions:

There exists a set of all world artefacts:  $\mathcal{U}$ . (3.1)

There exists a set of interesting artefacts:  $\mathcal{I}$ . (3.2)

The set of interesting artefacts is a subset of the set of all world artefacts:

$$\mathcal{I} \subseteq \mathcal{U} \quad (3.3)$$

The set of interesting artefacts,  $\mathcal{I}$ , can be intensionally defined as:

$$\mathcal{I} = \{x \in \mathcal{U} : \mathbb{I}(x)\} \quad (3.4)$$

where the condition  $\mathbb{I}$ , denotes interest. Section 3.2.2 will expand on the concept of interest conditions.



3.2.2 Interest Conditions

$\mathbb{I}(x)$  is an interest condition, which is analogous to an interest statement as introduced in Section 3.1.1.1.2.  $\mathbb{I}(x)$  is essentially a test which will indicate whether or not an artefact ( $x$ ) is interesting to us. For examples of such tests, consider the definitions of  $\mathbb{I}(x)$  in Table 3.1, and their corresponding illustrations in Figure 3.3:

Table 3.1: Example Interest Conditions

$\mathbb{I}(x)$	Description of Set $\{x \in \mathcal{U} : \mathbb{I}(x)\}$	Grey Items in Figure 3.3
$x$ is a square	the set of artefacts that are squares	(a)
$x$ is dotted	the set of artefacts that are dotted	(b)

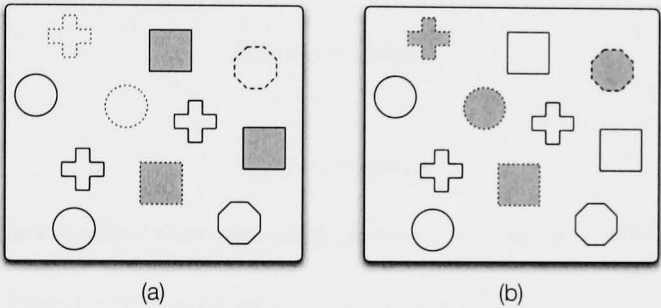


Figure 3.3: Interesting Artefacts

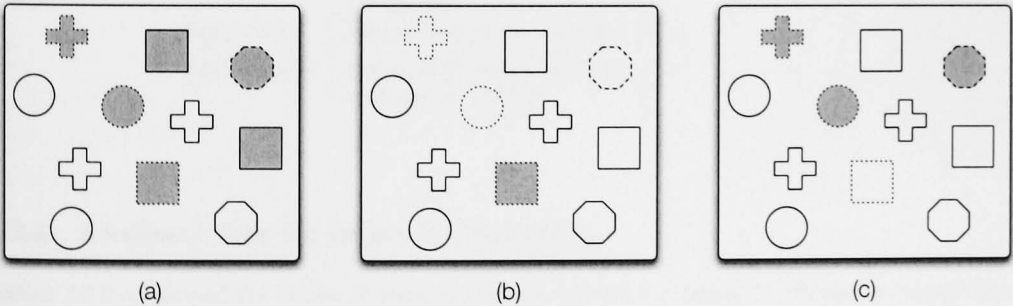


Figure 3.4: More Interesting Artefacts

### 3.2.3 Combining Interest Conditions

The construction of  $\mathcal{I}$  does not have to consist of just one interest statement as Section 3.2.1 might suggest. Multiple conditions may be used connected with the standard logic operators as presented in Table 3.2.

Table 3.2: The Standard Logical Operators

Operator	Symbol
And	$\wedge$
Or	$\vee$
Not	$\neg$

If we define the following conditions:

$$\mathbb{A}(x) = x \text{ is a square} \quad (3.5)$$

$$\mathbb{B}(x) = x \text{ is dotted} \quad (3.6)$$

It is then possible to combine them using logical operators as presented in Table 3.3.

Table 3.3: Combining Interest Statements with Logical Operators

Condition	Description of Set	Grey Items in Figure 3.4
$\mathbb{B}(x) \wedge \mathbb{A}(x)$	the set of all dotted artefacts and all artefacts that are squares	(a)
$\mathbb{B}(x) \vee \mathbb{A}(x)$	the set of all dotted squares	(b)
$\mathbb{B}(x) \wedge \neg \mathbb{A}(x)$	the set of all dotted artefacts that are not squares	(c)

### 3.2.4 Auxiliary Sets for Interest Conditions

Section 3.2.3 introduced the notion of creating complex interest conditions by combining simpler conditions with the standard logic operators. The examples shown were simple for pedagogic purposes, however, it is possible that these conditions can become extremely complex in certain circumstances. For example, if we briefly return to the example of a military simulation, a seemingly simple interest statement such as “*I’m interested in enemy soldiers*” could turn out to be extremely complicated. We might have to start considering combining conditions such as “*in line of sight*”, and “*in range of*”, etc.

And, of course, a statement such as “*in line of sight*” is itself not trivial, especially when we are reasoning at the level of artefact attributes.

Complicated conditions are difficult to write, and it would be easy to unknowingly inject errors. We need to have the ability to abstract away from the level of artefact attributes, thus giving us the ability to reason in terms of high-level concepts such as “*in line of sight*”, and “*in range of*”. This would allow us to create conditions such as “*I am interested in artefacts that are both in line of sight and within range*” This section explores the possibility of creating useful sets of artefacts specifically for this purpose. Section 3.2.4.1 will introduce the concept of creating sets of artefacts using the information within the virtual environment, and Section 3.2.4.2 will talk about creating supplementary sets of artefacts which add new information into the environment. Section 3.2.4.3 will introduce spatial sets as a specific example of auxiliary sets.

### 3.2.4.1 Derived Sets

In Section 3.2.1 we introduced two example interest conditions: the set of artefacts that are squares, and the set of artefacts that are dotted. We showed how these conditions could be used to construct values of  $\mathcal{I}$ , the set of interesting artefacts. However, we could use also these conditions to define a particular type of auxiliary set which is generated using the the condition  $\mathbb{D}(x)$ , a derived set. A derived set is a defined subset of  $\mathcal{U}$ , the set of all world artefacts. Consider the derived sets presented in Table 3.4.

Table 3.4: Derived Sets

Derived Set	$\mathbb{D}(x)$
SQUARES	$x$ is a square
DOTTED	$x$ is dotted

In Section 3.2.3 we described combinational conditions such as  $(x \text{ is dotted}) \vee (x \text{ is a square})$ . Using the derived sets introduced above, we could rewrite our condition as follows:  $(x \in \text{DOTTED}) \cup (x \in \text{SQUARES})$ . Derived sets can be combined with the following standard set operators:

The following illustrates the mapping between conditions using combinations, and conditions using derived sets as presented in Table 3.6.

Derived sets provide a mechanism for abstracting away from the level of artefact attributes. They can be seen as re-usable building blocks for constructing interest conditions.

### 3.2.4.2 Supplementary Sets

Section 3.2.4.1 described the two sets SQUARES and DOTTED which were derived from information obtained from current artefact attributes. In addition to deriving sets as described in Section 3.2.4.1, it is also

Table 3.5: The Standard Set Operators

Operator	Symbol
InterSection	$\cap$
Union	$\cup$
Superset	$\supseteq$
Subset	$\subseteq$
Proper Superset	$\supset$
Proper Subset	$\subset$
Member of	$\in$
Not a Member of	$\notin$

Table 3.6: The Mapping between Condition Combinations and Derived Sets

$\mathbb{I}(x)$ using combined conditions	$\mathbb{I}(x)$ using derived sets
$(x \text{ is dotted}) \vee (x \text{ is a square})$	$(x \in \text{DOTTED}) \cup (x \in \text{SQUARES})$
$(x \text{ is dotted}) \wedge (x \text{ is a square})$	$(x \in \text{DOTTED}) \cap (x \in \text{SQUARES})$
$(x \text{ is dotted}) \wedge (x \text{ is not a square})$	$(x \in \text{DOTTED}) \cap (x \notin \text{SQUARES})$

useful to inject information through the inclusion of supplementary sets.

Supplementary sets are sets of virtual artefacts which can be used within the construction of interest conditions. In order to further understand virtual artefacts, consider the concept of locales as introduced in Section 2.2.4.2. Locales can be seen to be artefacts that represent a region within the virtual environment. As this artefact does not actually ‘exist’ it is defined as a virtual artefact. If we assume we have a supplementary set of locales, `LOCALES`, we could create interest conditions such as the following:

$$\mathbb{I}(x) = x \text{ is within any of the locales within the set } \text{LOCALES} \quad (3.7)$$

Where the relationship “*within*” is further defined in Section 3.2.4.3. Supplementary sets provide a mechanism for inserting artefacts into the virtual environment specifically for the purpose of reasoning about interest. The virtual artefacts within supplementary sets contain state which is preserved, rather than state which is derived. These artefacts wouldn’t be explicitly visible, or available for interaction within the virtual environment, however their presence may be detectable through implicit means. The concept of virtual artefacts is similar to NPSNET’s concept of ghost artefacts[72].

### 3.2.4.3 An Example: Spatial Sets

Derived and supplementary sets provide useful building blocks for constructing interest conditions. Section 3.2.4.2 introduced locales as one use of supplementary sets. Locales can also be described using derived sets. This section will expand upon the concept of locales and spatial sets in order to further

illustrate the usefulness of derived and supplementary sets. We will also see that derived and supplementary sets are not entirely orthogonal concepts, and discuss any similarities found.

Figure 3.5 shows a simple virtual environment separated into nine spatial areas ( $A'..I'$ ) which are represented as virtual artefacts. Each of these areas can be used to define a set of artefacts - the set of artefacts within each area. The derived set of artefacts representing the set of all artefacts within  $A'$  can be intensionally written as follows:

$$\text{WITHIN-}A = \{x \in \mathcal{U} : x \text{ is within } A'\} \quad (3.8)$$

Clearly, the set  $\text{WITHIN-}A$  contains artefact 1, however, whether it also contains artefact 4 is not so clear. For the answer to this problem it is necessary to clarify the definition of the term "*within*". There are a number of ways of clarifying this. For example, we could say an artefact is only within an area if it is entirely contained within the boundaries of the area, e.g. artefact 1 is entirely within the boundaries of area  $A'$ . Another option would be to define the midpoint for an artefact, and then calculate the co-ordinates of that midpoint. As long as we assume that we can always determine whether a co-ordinate lies within an area or not, then there are no issues of uncertainty. Neither of the two options are *correct*, and indeed there are more methods available. An appropriate definition for the circumstances must be defined by the designers, within the context of the virtual environment.

Examples of spatial operators are presented in Table 3.7.

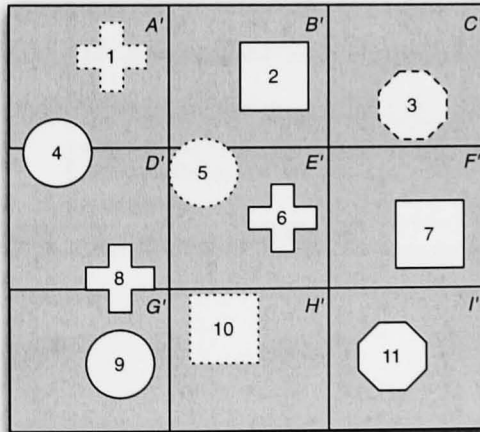


Figure 3.5: Spatial Sets within a Virtual Environment

It is important to consider that the spatial sets do not necessarily have to be uniform, or tessellate, or even cover the entire virtual environment as Figure 3.5 suggests. As, in this case, they are represented

Table 3.7: Spatial Operators

Operator	Description
DISJOINT	the boundaries and interiors do not intersect
TOUCH	the boundaries intersect but the interiors do not intersect
OVERLAPBDYDISJOINT	the interior of one object intersects the boundary and interior of the other object, but the two boundaries do not intersect. This relationship occurs, for example, when a line originates outside a polygon and ends inside that polygon
OVERLAPBDYINTERSECT	the boundaries and interiors of the two objects intersect
EQUAL	the two objects have the same boundary and interior
CONTAINS	the interior and boundary of one object is completely contained in the interior of the other object
COVERS	the interior of one object is completely contained in the interior of the other object and their boundaries intersect
INSIDE	the opposite of CONTAINS. A INSIDE B implies B CONTAINS A
COVEREDBY	the opposite of COVERS. A COVEREDBY B implies B COVERS A
ON	the interior and boundary of one object is on the boundary of the other object (and the second object covers the first object). This relationship occurs, for example, when a line is on the boundary of a polygon
ANYINTERACT	the objects are non-disjoint

using virtual artefacts, they can be of any shape and in any position. Figure 3.6 illustrates this.

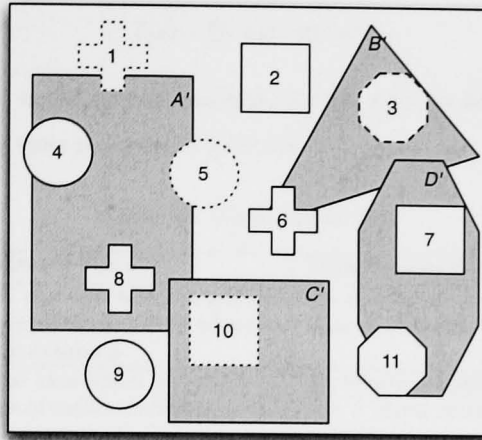


Figure 3.6: spatial Sets can be of any shape or orientation

spatial sets are useful for a variety of purposes. They can be used to represent areas or locales within a virtual environment. They can also be used to define viewing frustums, artefact auras, and focii. These are all areas which may be associated with particular artefacts. For example, in Figure 3.6, spatial set  $B'$  may represent the viewing frustum for artefact 6. Virtual artefacts are similar to the concept of the parallel virtual world introduced by Oliveira for the VELVET system[32].

As we have seen, spatial sets can be created using a combination of derived sets and supplementary sets. However it is possible to recreate certain aspects of supplementary sets using derived sets. For example, it would be possible to describe the spatial sets within Figure 3.5 through the definition of the derived set, i.e. the definition of set  $A'$  would be included within the intensional definition of `WITHIN-A` rather than relying on the existence of a supplementary artefact representing  $A'$ . Where there is the choice of using derived or supplementary sets to define the condition, care should be taken to choose the most appropriate type in terms of the context of the decision.

### 3.2.5 Relative Interests

So far, we have only really considered interest from a global perspective. As introduced in Section 3.1.3, external users are often associated with an artefact within the virtual environment. If this is the case, then the user may wish to have an interest which is relative to an associated artefact.

Consider that there exists an artefact or avatar  $a$  which may be associated with a particular user. Our condition may then take in two parameters:  $x$  being the artefact the user wishes to determine the interest of, and  $a$  the artefact our interest is relative to. Our interest condition can therefore be written

as  $\mathbb{I}(x, a)$ , and the set builder for interest relative to artefacts is:

$$\mathcal{I}(a) = \{x \in \mathcal{U} : \mathbb{I}(x, a)\} \quad (3.9)$$

Given the presence of  $a$  in our interest condition, we can use it to define some useful spatial sets relative to artefacts such as those presented in Table 3.8.

Table 3.8: Useful Spatial Sets

Set	Condition: $\mathbb{P}(x, a)$	Description of set
AURA	$x$ is in the area which is defined by a circle of radius 10 which is centred around $a$	An area which bounds the presence of an artefact
NIMBUS	$x$ is the area which is defined by a circle of radius 30 which is centred around $a$	An area which represents how visible a given artefact is to other artefacts
FOCUS	$x$ is a cone which is defined by a radius of 10 and a height of 30, the tip of which is centred around $a$	An area which represents the range of visibility of a given artefact in order to see other artefacts

### 3.3 Example Interest Statements

This section aims to provide some general interest statements using the model introduced in this chapter. The aim is to illustrate that the interest management model can represent the range of interest management techniques categorised in Section 2.2.4. Implementations of these will be discussed in Chapter 4.

#### 3.3.1 Locales

As introduced in Section 2.2.4.2, locales are spatial regions within spatial virtual environments. Given that the virtual environment is spatial, each artefact will have associated co-ordinate attributes. Using these co-ordinates it is possible to determine whether a particular artefact is within a particular locale. As we saw in Section 3.2.4.3, it is possible to use these locales to reason about the interest of a particular artefact using set builders such as the one described in Equation 3.8, and repeated here:

$$\mathcal{I} = \{x \in \mathcal{U} : x \text{ is within } A'\} \quad (3.10)$$



### 3.3.2 Relative Locales

As introduced in Section 2.2.3.7, and further described in Section 3.2.5, locales (and interests in general) may be relative to a particular artefact. An example of this is a viewing frustum: a locale (typically represented by a sphere, or cone) which represents the area that an associated artefact can see. Again, this viewing frustum could be represented using a virtual artefact, and the set builder for an interest represented with relative locales is :

$$\mathcal{I} = \{x \in \mathcal{U} : x \text{ is within } A' \wedge A' \text{ is associated with } a\} \quad (3.11)$$

### 3.3.3 Interacting Locales

As introduced in Section 2.2.4.3, the interaction of locales can be used to reason about interest. For example, when an artefact A's aura collides with artefact B's aura, artefact A can be said to be aware of artefact B.

$$\mathcal{I}(a) = \{x \in \mathcal{U} : \text{the aura associated with } a \text{ overlaps the aura associated with } x\} \quad (3.12)$$

Equation 3.12 can be more generally stated as the following:

$$\mathcal{I}(a) = \{x \in \mathcal{U} : \text{the locale associated with } a \text{ interacts with the locale associated with } x\} \quad (3.13)$$

where the interaction relationships are the standard spatial set relationships<sup>1</sup>: *within*, *contains*, *overlaps*, and *touches*[93].

### 3.3.4 Categories

As introduced in Section 2.2.4.1, using a class-based system to categorise the virtual environment is an easy way to reason about interest. Section 3.2.4.1 introduced the concept of derived auxiliary sets, which is essentially a mechanism for creating classes of artefacts. For example, we could create an auxiliary set which is derived by looking at the colour attribute of all artefacts, and selecting the ones that are red (*all red artefacts*), and use that set within our set builder:

$$\mathcal{I} = \{x \in \mathcal{U} : x \in \text{CATEGORY}\} \quad (3.14)$$

where CATEGORY represents any derived auxiliary set.

---

<sup>1</sup>Note that the spatial model introduced in the MASSIVE systems doesn't typically use standard spatial set relationships such as overlap e.g. *sample field*.

### 3.3.5 Combinations

A benefit of this approach is that the techniques described above can be combined in arbitrary ways to create more complex expressions. For example, if we want to be interested in all red artefacts that are within our viewing frustum:

$$\mathcal{I}(a) = \{x \in \mathcal{U} : (x \in \text{RED}) \wedge (x \text{ is within } A') \wedge (A' \text{ is the viewing frustum of } a)\} \quad (3.15)$$

Describing interests using the conceptual model in this way gives us a more flexible mechanism for representing interest within virtual environments than previously possible.

## 3.4 Constraints and Conflicts

So far, this chapter has introduced and described interest statements. The motivation for these interest statements being the ability to have a flexible way of representing interest. However, this flexibility has a potential to interfere with the purpose of the environment. In Section 2.1.3 we introduced the range of current virtual environment applications. Each of these applications has a purpose, and each of those purposes may impose one or more constraints on the choice of interest. This section will explore this in greater detail.

Section 3.4.1 will introduce the concept of constraints on interests for specific virtual environments. These constraints will be introduced with relevant examples, and solutions will be given.

### 3.4.1 Interests for Specific Virtual Environments

When virtual environments are designed and implemented, it is usually for a specific purpose, for example: collaboration, simulation, or research. This section looks at the potential constraints that a virtual environment may place on interest, and conflicts that may occur between the users interest and that of the environment.

#### 3.4.1.1 Relative Visibility

If we assume that each user has control of their interests, then the visibility of the artefacts is relative to each user. This means that each user may have a different view of the same set of artefacts or scene, which could lead to potential problems. For example, an interest in cups, and not tables may lead to a view of cups seemingly floating in the air. This is further illustrated in Figure 3.7. Figure 3.7(a) shows a simple virtual environment with 6 users. We are only concerned with the users *A* and *B*. User *A* is interested in all users that aren't dotted, user *B* is interested in all users within the virtual artefact *V'*. Figure 3.7(b) shows the artefacts that are within user *A*'s interest set, Figure 3.7(c) shows the artefacts

which are within user  $B$ 's interest set. As we can see from the illustrations, user  $B$  can see user  $A$ , but user  $A$  cannot see user  $B$ . In this case, visibility is not a commutative relationship. If the goal of the virtual environment is to facilitate communication between nearby users, then this situation clearly poses a problem: user  $A$  cannot see user  $B$  to know that there's someone to communicate with. The concept of relative visibility is similar to the degree of blindness concept introduced by Oliveira and Georganas[32].

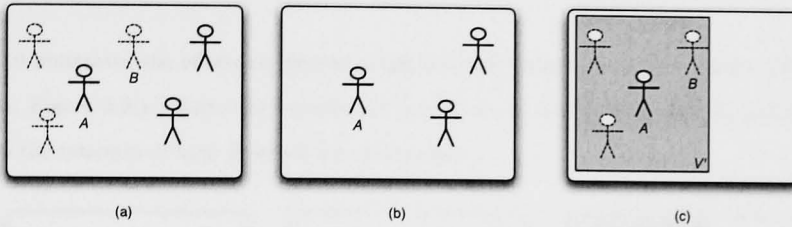


Figure 3.7: Relative Visibility

### 3.4.1.2 Constraints

The specific purpose of the virtual environment will introduce constraints on what should and should not be interesting. As we saw in Section 3.4.1.1, the interest of user  $A$  in Figure 3.7 conflicted with the purpose of the environment. Also, consider the situation of the virtual environment being a war simulation. If the user is a soldier on the ground, then the user *needs* to be interested with everything within line of sight. Alternatively, if the virtual environment is a virtual lecture, and the user is a member of the audience, then the user *needs* to be interested with the speaker.

Figure 3.8 illustrates two virtual artefacts,  $A'$  and  $B'$  which represent the viewing frustum of users  $A$  and  $B$  respectively.

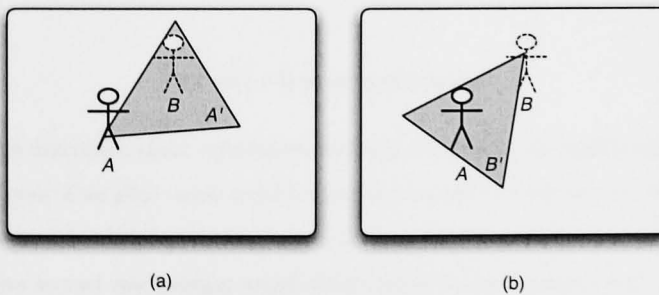


Figure 3.8: Constraints Introduced

Using the two virtual artefacts  $A'$  and  $B'$  introduced in Figure 3.8, it is possible to rewrite the interest

statements of users  $A$  and  $B$  as presented in Table 3.9.

Table 3.9: Dealing with Constraints

User	Previous Interest $\mathbb{I}(x)$	Updated Interest $\mathbb{I}(x)$
$A$	$\neg x$ is dotted	$\neg x$ is dotted $\vee x$ is within $A'$
$B$	$\neg x$ is within $V'$	$\neg x$ is within $V' \vee x$ is within $B'$

Figure 3.9 illustrates the effects of interest using the new updated interests. Figure 3.9(a) shows all the artefacts, Figure 3.9(b) shows the interests of user  $A$  which now include user  $B$ , and finally, Figure 3.9(c) shows the interests of user  $B$  which are unchanged.

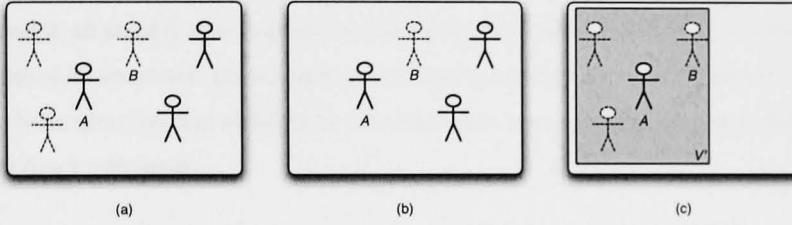


Figure 3.9: Updated Interests for Users  $A$  and  $B$

For a virtual environment to meet its objectives, there must be minimum requirement,  $\mathcal{O}$ , of what a user must be aware of. This minimum requirement is essentially a subset of all world artefacts:

$$\mathcal{O} \subseteq \mathcal{U} \quad (3.16)$$

Each user will have a set  $o$  associated with them, and the condition,  $\mathbb{O}(x, a)$ , that describes this set can be written as:

$$\mathbb{O}(x, a) = x \text{ is essential for artefact } a \quad (3.17)$$

As Section 2.3.4 described, these constraints are hard-coded into the implementation of the virtual environment. However, if we allow users to have complete control of their interest, then we need to take these constraints into consideration in the process of describing  $\mathcal{I}$ , our interest set. For example, if we have a collaborative virtual environment which allows users to communicate with each other, then we need to represent that requirement as a constraint, and enforce it. As seen in the example above, this enforcement can be achieved by combining the user's interests ( $\mathcal{I}$ ), and the objectives of the virtual environment ( $\mathcal{O}$ ) as follows:

$$\mathbb{E}(x, a) = \mathbb{I}(x, a) \wedge \mathbb{O}(x, a) \quad (3.18)$$

### 3.4.1.3 Conflicts

In Section 3.4.1.2, we described how the environment may introduce a set of constraints, and showed how to alter our interests to cater for them. This process turned out to be simply combining the interests of the user, and the constraints of the virtual environment, as described in Equation 3.18. This section will describe how these constraints and interests may actually conflict, and present mechanisms for conflict resolution.

Consider Figure 3.10(a) which has four artefacts: three people, and a wall. Using either an associated viewing frustum virtual artefact as in Figure 3.10(b), or a simple categorisation based interest condition (I'm interested in all people), user *A* would be able to see both users *B* and *C*. This however, may not be appropriate in all situations. For example, user *C* may be attempting to hide from user *A* behind the wall. It may be an objective that visibility be calculated more appropriately than using these potentially naïve spatial based techniques.

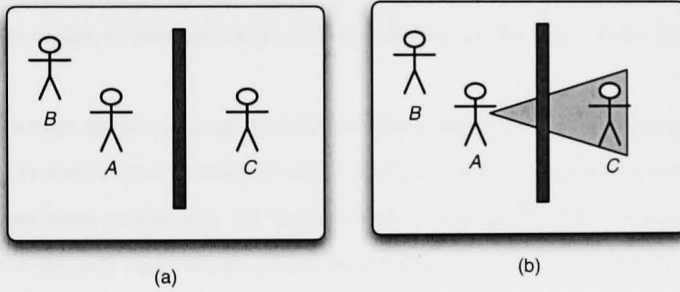


Figure 3.10: Blocked Visibility

There are many techniques for calculating the visibility of artefacts in virtual environments, and all are essentially algorithms based on the current values of artefact attributes. Let us consider that we have implemented such a technique<sup>2</sup>:  $\mathbb{V}(x, a)$  which defines the condition “*is a visible to x?*”. We can define a relative set ( $\mathcal{R}$ ) of all artefacts visible to  $x$  as follows:

$$\mathcal{R} = \{x \in \mathcal{U} : \mathbb{V}(x, a)\} \quad (3.19)$$

Using this new notion of visibility, we can avoid any conflicts between the goals of the virtual environment, and the interests of the user by combining these interests as follows<sup>3</sup>:

<sup>2</sup>Such as the approach by Teller and Sequin[110].

<sup>3</sup>Note that the calculation of  $\mathbb{V}(x, a)$  is dependent on the algorithm used, and has the potential to be extremely complex

$$\mathcal{R} = \{x \in \mathcal{U} : \mathbf{I}(x, a) \wedge \mathbf{V}(x, a)\} \quad (3.20)$$

Here we are saying that we are only interested in things which are interesting, combined with the things that are visible. Therefore, if something is not visible, we can not be interested in it – even if we want to be.

### 3.4.2 Separation of Concerns

Section 3.4 introduced the notion of constraints and conflicts of interest within a virtual environment. This section will generalise these constraints into two types: *positive* and *negative* enforcement, and two subjects: the *user*, and the *simulation*. Through this generalisation we see how we can start to harness the power of dynamic, expressive interest statements.

In the following Sections, the user is defined as the external entity interacting with the virtual environment. As explained in Section 3.1.3, a user is typically associated with an artefact (usually called an avatar). What we are concerned with here is that the user may have interests within the virtual environment. Also, any associations with artefacts would allow us to express relative interests (as introduced in Section 3.2.5). There may be more than one user in the virtual environment. However we are only concerned with the notion of one particular user; the identity of the user, or the existence of others is not important.

In addition to a user, there is also the simulation. The simulation can be conceptually regarded as a user representing the virtual environment. However, there is not typically such an entity, but it is useful to conceptualise one when considering the requirements or interests of the virtual environment. These issues are similar to the role based access control mechanisms described by Brunton et al.[20]

#### 3.4.2.1 User Interests: Positive Enforcement

The main subject of this chapter has been the definition of interests for a particular user. The implicit assumption being that we have been defining the things that the user is interested in. A user's positive interests are the explicit definition of this implicit assumption. It is an interest statement describing the set of artefacts that the user is positively interested in, i.e. the things the user wants to see.

#### 3.4.2.2 User Interests: Negative Enforcement

As discussed in Section 3.4, we also need to describe a set of artefacts that are not interesting. A user's set of negative interests is an interest statement describing the set of artefacts the user is not interested in, i.e. the things the user does not want to see. Negative interests may be useful to express in addition and depend on all the artefacts within the virtual environment.

to positive interests where there may be a set of optional interests that the user might want to see. By expressing a set of negative interests, this optional set can be appropriately pruned. However, there may also be interests that might not be optional, but enforced by the simulation, which are discussed in the following sections.

#### 3.4.2.3 Simulation Interests: Positive Enforcement

The simulation may have a set of goals or objectives. These would be represented by an interest statement defining the set of artefacts that a particular user *must* be interested in.

#### 3.4.2.4 Simulation Interests: Negative Enforcement

In opposition to positive enforcement, the simulation may have an interest statement defining the set of artefacts that a particular user is *not* allowed to be interested in.

### 3.4.3 Combining Interests

In order to make use of the various concerns, we need a method of combining them. For this to occur, we must assume a priority: that the simulation's concerns are more important than the users. Using this assumption we can combine these statements as follows:

$$\text{Interesting Artefacts} = ((\text{UPOS} - \text{UNEG}) \cup \text{SPOS}) - \text{SNEG} \quad (3.21)$$

Where UPOS, UNEG, SPOS and SNEG are defined in Table 3.10.

Table 3.10: Combining Concerns of Interest

Abbreviation	Full Name of Set
UPOS	The user's positive enforcements
UNEG	The user's negative enforcements
SPOS	The simulations's positive enforcements
SNEG	The simulations's negative enforcements

## 3.5 Summary

This chapter has defined the axioms of a virtual environment necessary to reason about interest management. Building upon these axioms, a categorisation system was defined which was then used in the construction of interest statements. These interest statements were defined using simple set theory, and were mapped onto the categories of interest statement introduced in Section 2.2.4. Following this, the

issues of constraints and concerns were looked at, with a solution for dealing with conflicts of interests offered. The next chapter will introduce an implementation of these ideas to reify the concepts introduced.



## Chapter 4

# Virtual Environment Axioms: A Proof of Concept

Chapter 3 introduced a framework capable of representing dynamic interest management. It defined a set of axioms for a conceptual model of a virtual environment (Section 3.1), and then introduced **set theory** as a mechanism for representing statements of interest (Section 3.2) which built on top of these axioms. This chapter will explore design and implementation decisions of the axioms presented in Section 3.1.2, and present a reference implementation. This implementation will be a foundation to present the work on interest management in the following chapters.

Section 4.1 will revisit the virtual environment axioms presented in Section 3.1.2 and describe various implementation methods. Section 4.2 will discuss the process of designing the data structure required to support the implementation of a simple virtual environment. Section 4.3 will describe the implementation decisions made. Finally, Section 4.4 describes the process of implementing the virtual environment itself.

### 4.1 The Axioms Revisited

Section 3.1.2 introduced artefacts, time, events and processes as core components of a virtual environment. This section will look at these components in turn and introduce designs which will be implemented in order to provide the foundation for a dynamic interest management framework.

#### 4.1.1 Artefacts

Section 3.1.2.2 introduced artefacts as *“individual units of the virtual environment”* where *“each artefact consists of one or more attributes”*. A simple list of the attributes that could be used to compose such an artefact was as follows:

- unique identifier,
- colour,

- x co-ordinate,
- y co-ordinate,
- z co-ordinate,
- geometric description.

Two possible ways of implementing such artefacts are using a language structure such as an object and using a relational database structure such as a table.

The following two sections will explore both these possibilities and show that they are not necessarily mutually exclusive designs, but are able to work together well to yield good benefits.

#### 4.1.1.1 Artefacts as Objects

If we are to consider that each artefact has each of these attributes, then one way of realising an implementation is to map each artefact onto an object in an object-oriented language. Representing an artefact within a programmatic entity such as an object allows us to use the programming language to represent both the data that the object consists of, and perform operations on that information. For example, each object could store an object's set of attributes as instance data, and provide methods (both class and instance) to manipulate that data. A very simple example of such an object could be described as follows:

```
class Artefact
  attr_accessor :id, :colour, :x_coord, :y_coord, :z_coord, :shape
end
```

This example is in Ruby<sup>1</sup>, and shows the attributes that each new instance of class `Artefact` is born with. Given this class, we can create new instances and edit their attributes as follows:

```
cube = Artefact.new
cube.id = 1
cube.colour = "red"
cube.x_coord = 10
cube.y_coord = 11
cube.z_coord = 12
cube.shape = "cube"
```

---

<sup>1</sup>The method `attr_accessor` is one of many examples of metaprogramming techniques found within Ruby, and when interpreted will generate instance variables and accessor methods (getter and setter methods in Java lingo) for all the parameters that it is passed.

The benefit of this approach is the ease by which behaviour can be added to the artefacts where necessary. In addition to storing attributes, O-O objects are also capable of storing associated behaviour. In Ruby this is at both the class level, and also at the individual instance level. This gives the potential of representing behaviour unique to each artefact.

#### 4.1.1.2 Artefacts as Table Rows

Another way of realising an implementation of artefacts is using database tables. Storing data within a database provides the ability to persist the data, and allows for powerful querying over large sets of information using already defined languages such as SQL. Each table within a database consists of rows and columns. In the context of our artefact, we can consider the attributes of an artefact to be represented by the columns of the database table. Each row of the table would refer to a different artefact. The id attribute of the artefact would make a very reasonable primary key.

An ActiveRecord database migration for creating such a table could look as follows:

```
create_table :artefacts do |t|
  t.column :id,      :integer
  t.column :shape,   :string
  t.column :x_coord, :float
  t.column :y_coord, :float
  t.column :z_coord, :float
  t.column :colour,  :string
end
```

The benefits of this approach are automatic persistence and power of SQL as a querying tool. Amongst other abilities, SQL is capable of searching, manipulating, and merging information within a database. One of the goals of this thesis is to create a language for interest statements. With the virtual environment information stored in a relational database, SQL proves to be an excellent candidate for implementing such a language.

#### 4.1.1.3 Combining Objects and Tables: Object Relational Mapping

The two methods of implementing an artefact as presented above are by no means exclusive. In fact, the two methods compliment each other remarkably well. Combining programmatic objects with database structures is a common pattern found in application development - particularly applications found on the web. Object relational mappers (ORMs) are tools or libraries that are concerned with exactly this task.

The combination of two approaches to implementing the artefact described in Sections 4.1.1.1 and 4.1.1.2 is in fact the active record pattern proposed by Martin Fowler[36]. Active record is a simple

and intuitive design pattern that can be found in many enterprise applications. It is an approach to object relational mapping, whereby objects in object oriented programming are directly mapped to rows in particular tables in the database. This idea is taken a little further by ActiveRecord (a Ruby implementation of the active record pattern) which maps the object class name to the database table name. This naming mapping defaults to the object being the singular form of the concept being modelled (Artefact), and the database table being the plural form of the concept (artefacts). This therefore reduces the amount of configuration that is usually required with most ORM tools to describe exactly how the objects map to the database structure.

Using a combination of database structure (artefacts as table rows), and an O-O class structure (mapping objects to table rows), we have a powerful combination with which to define, store, and use artefacts. The artefacts can be stored in the database, and used as objects. The objects can allow us to define artefact behaviour where necessary and the database can provide persistence and the ability to use SQL to generate interesting sets of artefacts.

#### 4.1.2 Time

In the context of implementing a virtual environment to support an interest management framework, having a numeric representation of time is only useful if we wish to use that concept whilst describing interests (i.e. I'm interested in food in the morning, and beds at night). For the purposes of maintaining simplicity, we shall assume that a numeric representation of time is not a requirement. Instead, we shall let the concept of time refer to the existence of a current state, and the ability to change that state into a new state. If we are to store our data within a database, this is easily achieved by issuing UPDATE commands to the database in order to change the state, and SELECT commands to refer to the current state.

#### 4.1.3 Events

In Section 4.1.2 we described the need for successive versions of state within our virtual environment. A change in this state is caused by an event. In Section 3.1.2.3 we introduced events as "an alteration of one or more attributes of one or more artefacts at a particular time". Given that in our case, time refers to the current state, we are therefore interested in the ability to alter one or more attributes, of one or more artefacts. Section 4.1.2 also referred to database UPDATE commands as being able to alter state. Provided that our state is stored within a database, the UPDATE command allows us to update one or more attributes of one or more artefacts.

Here is an example in SQL of an update to the artefact that has an id of 2:

```
UPDATE artefacts SET
```

```

"x_coord"      = 32.0,
"y_coord"      = 4.0,
"z_coord"      = 0.0,
"shape"        = 'cube',
"colour"       = 'red',
"transparency" = 0.5
WHERE id = 2

```

In addition, this SQL example can be mapped to object-oriented style syntax by ActiveRecord, to the following:

```

Artefact.find(2).update_attributes(
    :x_coord => 32.0,
    :y_coord => 4.0,
    :z_coord => 0.0,
    :shape   => 'cube',
    :colour  => 'red',
    :transparency => 0.5
)

```

ActiveRecord will convert the `update_attributes` call to the equivalent of the SQL given above.

#### 4.1.4 Processes

Section 3.1.2.5 introduces processes as *“a logical grouping of events that are inter-linked through a particular relationship”*. These could be represented by standard Ruby blocks by ActiveRecord. However, as is evident by their absence in the rest of this Chapter, processes are not an essential part of the supporting infrastructure for the interest management framework, and will be left for further discussion in Chapter 8.

## 4.2 Data Design

As described in Section 4.1, Each of the axioms (time, events, artefacts, and attributes) introduced in the conceptual virtual environment in Section 3.1 easily map onto a database and object-oriented equivalent as described in Table 4.1

In order to keep the design simple, one table named `artefacts`, was used to represent the information for all artefacts. Each individual artefact maps to a particular row of the table, and the attributes of an

artefact map to the columns of the table. This is according to the ActiveRecord pattern. In the initial design, the attributes as presented in Table 4.2 were used to collectively represent an artefact.

Most of these attributes should seem relatively straight forward, except perhaps for transparency. Transparency is a floating point value between 0 and 1 where 0 is opaque, and 1 is transparent (invisible). A transparency value of 0.5 would indicate that the artefact is half transparent. Transparency will allow us to view artefacts that might be contained within artefacts.

## 4.3 Implementation Decisions

The section will describe the various factors considered before and during the design and implementation of the virtual environment conceptual model.

### 4.3.1 Data Storage Technology

The virtual environment needs some kind of storage mechanism for storing the system's data. On the simplest level, these data are the artefacts and attributes that the artefacts consist of. Section 4.1.1.3 introduced a relational database as a storage mechanism that supported the factors necessary to represent a virtual environment's data. This section will explore these factors in more detail, and present databases as an obvious choice given a full consideration of all the factors.

The data store needs to at least have the following factors which will be explored in the following Sections: persistence, support for set structures, support for querying, and support for spatial queries. Section 4.3.1.5 will describe the final implementation choices.

#### 4.3.1.1 Persistence

The storage mechanism needs some way of surviving system crashes, or simple system reboots. One way of achieving this is to represent the information in such a way that it can be stored on some kind of persistent storage technology. There are many options which facilitate this requirement. For example, the data could be represented as files on a file system such as ZFS, HFS+, or NTFS which in turn is stored on a persistent storage technology such as NAND-type flash memory data storage devices, or hard

Table 4.1: Mapping between Conceptual Model Term, Database Term, and Object-Oriented Term

Conceptual Model Term	Database Term	O-O Term
Time	the ability to alter data	the ability to edit object attributes
Event	an UPDATE on a database table	an update_attributes method
Artefact	a database table row.	an instance of class Artefact
Attribute	a database table column	instance data for Artefact instances

Table 4.2: Attributes for the Initial Design

Attribute	Data type	Example Box	Example Sphere
id	integer	1	1
shape	string	box	sphere
width	float	5.0	nil
height	float	10.0	nil
length	float	15.0	nil
radius	float	nil	10.0
x_coord	float	1.0	2.0
y_coord	float	2.0	4.0
z_coord	float	3.0	6.0
colour	string	red	blue
transparency	float	0.5	0.0

disks.

#### 4.3.1.2 Support for Set Structures

In order to support the ideas on interest presented in Chapter 3, the data storage mechanism needs to have support for representing sets of data. We need to be able to store sets of artefacts which are in turn sets of attributes. This is possible using purpose made data structure, or using a more generic relational database.

#### 4.3.1.3 Support for Querying

The motivation for storing the information in a set structure, is that (as was illustrated in Chapter 3) it is possible to use set theory to pull out subsets of that information - potentially interesting subsets. Therefore a language that allows you to represent and execute snippets of set theory over a set structure is necessary. Such a language is SQL which is the primary querying language in the relational database world.

#### 4.3.1.4 Support for Spatial Queries

If the virtual environment consists of spatial information, it is potentially very useful to make queries that can reason about spatial matters. This matter was discussed in Section 3.2.4.3. There already exist a number of libraries which support spatial queries, one of which is Oracle Spatial[93]. Oracle Spatial provides the ability to represent queries with keywords such as touch, contains, covers, inside and covered by. These keywords enable the query to reason about spatial relationships between artefacts, which would allow for interests with respect to locales and representational zones such as viewing frustums and auras.

#### 4.3.1.5 Implementation Choice

Clearly the relational database is an obvious implementation choice. Most available databases provide persistence, support for set structures, and support for querying out of the box. Some, such as Oracle, provide support for spatial queries too.

The original prototypes of the virtual environment implementation were built using Oracle 9i as the database. One of the main motivations for using this database was the built in support for spatial queries[93]. However, Oracle 9i is a very heavyweight database implementation, and needs to be deployed on its own server due to its demanding resource requirements. The current university network restrictions meant that all development had to occur on the same network, which constrained the locality of any development. It was important to be able to develop the implementation in the absence of network connectivity, and therefore a database which could run alongside the rest of the implementation on the primary development machine was required. Therefore, for the final implementation, MySQL[1] was used. MySQL does not support spatial queries, however the logic to deal with spatial relationships was moved up into the language layer (as will be discussed in Chapter (5)). This allowed the entire framework to run on one machine which helped the development, deployment and ongoing testing of the software system. Oracle Spatial is a very interesting extension to the Oracle database, and if further work takes place looking at potential optimisations (particularly for spatial calculations) this would be an excellent place to start (for further discussion see Chapter 6).

### 4.3.2 Development Methodology

Initial prototypes of the system suffered from a constant evolution and change of the specifications and requirements. This resulted in messy, poorly structured, and error-prone code. In order to manage this situation, the final implementation was built in a test-first manner using a set of executable specifications written in RSpec (see Section 2.5.2). RSpec follows a concept known as Behaviour Driven Development (BDD[11]) which is an evolution of the more practised Test Driven Development (TDD[30]) methods. TDD and BDD are discussed in the following sections (4.3.2.1 and 4.3.2.2).

#### 4.3.2.1 Test Driven Development

Test Driven Development follows a very simple development cycle:

- write a test (which should fail)
- write the code to make the test pass
- refactor the code



Following this cycle means that all the way through the development of the software, there is an evolving set of tests that you can constantly run to check that the system behaves as expected. Implementations of TDD frameworks are typically based upon the XUnit framework[10]. There exist many implementations in many different languages.

#### 4.3.2.2 Behaviour Driven Development

Behaviour Driven Development is essentially an evolution of TDD, particularly in terms of vocabulary. It is a term coined by Dan North, and attempts to move away from the concept of testing, and towards the practice of writing executable specifications of system behaviour. Specifications written in a BDD framework are typically far more readable than tests written using a TDD framework. This facilitates the process of verifying that the tests/specifications are correct in terms of describing the behaviour of the system.

There are currently not many BDD frameworks available, however RSpec, a BDD framework written for Ruby is remarkably robust and versatile. Using RSpec to develop the virtual environment resulted in much more stable, reliable, and trusted code-base. RSpec is discussed further in Section 2.5.2.

#### 4.3.3 Implementation Language

The original prototypes of the implementation were originally written entirely with Java. However, as the designs changed, the code-base gradually grew more unwieldy, and refactoring started to become more error prone. The final version of the prototype has been mostly written in Ruby, with one part of the original prototype remaining in Java. Various concepts found within Ruby, such as blocks (closures), meta-programming and dynamic typing have facilitated the reduction of the code-base by a factor of 2-3 times, resulting with far leaner and more elegant code.

##### 4.3.3.1 Supporting Libraries

The code was developed using a BDD (Behaviour Driven Development) approach using RSpec (see Section 2.5.2). This allowed the code to be tested at every iteration to ensure that it conformed to the specifications.

Ruby also has the ActiveRecord library which simplifies the process of communicating with the database, and managing schema changes.

## 4.4 Implementation

This section will describe the process of implementing the virtual environment system. Section 4.4.1 will describe the process of creating the database schema, Section 4.4.2 will describe the definition of the

Artefact class, and Section 4.4.3 will describe a sample virtual environment. Section 4.4.4 will introduce interest within this context, and then Sections 4.4.5, 4.4.6 and 4.4.7 will expand on the implementation, introducing a 3D viewing mechanism, a client and server architecture, and an incremental update message format.

#### 4.4.1 Creating the Database Schema

The Ruby ActiveRecord library makes the process of turning this design into a real database table remarkably simple. ActiveRecord provides a light abstraction above SQL, allowing you to define your database schema using a slightly more readable syntax. Migrations are part of a set of functionality internal to ActiveRecord and provide the ability to store changes in separate, ordered, files allowing for rolling forward and back along the schema history. However, at this stage we are only concerned with creating the artefacts table. The part of the migration of interest is as follows:

```
create_table :artefacts do |t|
  t.column :shape,      :string
  t.column :width,      :float
  t.column :height,     :float
  t.column :length,     :float
  t.column :x_coord,    :float
  t.column :y_coord,    :float
  t.column :z_coord,    :float
  t.column :radius,     :float
  t.column :colour,     :string
  t.column :transparency, :float
end
```

As can be seen, this migration is similar to the design presented in Table 4.2. One thing to notice is that there is no `id` column. This is due to the fact that by default ActiveRecord creates and manages this column for us. Executing this migration will result in the creation of a matching table within our database.

#### 4.4.2 Defining the Artefact Class

Using ActiveRecord as the object relational mapper makes defining the Artefact class easy:

```
class Artefact < ActiveRecord::Base
end
```

Simply inheriting from `ActiveRecord::Base` gives the `Artefact` class everything it needs. `ActiveRecord` is able to inspect the appropriate database table (artefacts being the plural of `Artefact`), and generate the necessary instance methods on the fly at run time. The artefact class, once defined, is able to create, read, update and delete artefact objects in an object oriented fashion.

#### 4.4.3 A Sample Virtual Environment

In order to test this implementation, it is necessary to create a very simple example virtual environment. Using the `yaml`<sup>2</sup> markup language, we can define the following fixture<sup>3</sup> containing some sample artefacts with which to populate our database:

```
sphere:
  shape: sphere
  radius: 2
  x_coord: 10
  y_coord: 15
  z_coord: 0
  colour: green
  transparency: 0.5
cylinder:
  shape: cylinder
  radius: 1
  height: 10
  x_coord: 10
  y_coord: 5
  z_coord: 0
  colour: turquoise
  transparency: 0.5
floor:
  shape: floor
  width: 100
  length: 50
  transparency: 0
box:
  shape: box
  height: 5
  width: 2
  length: 10
  x_coord: 50
  y_coord: 20
  z_coord: 0
  colour: red
  transparency: 0.5
```

This fixture defines four artefacts: a floor, a box, a sphere and a cylinder. Using `ActiveRecord`, we can import the data within this fixture straight into our database.

<sup>2</sup>see Section 2.5.1

<sup>3</sup>Fixtures are simply a set of predefined sample data which can be used to populate the database, and are typically used for testing.

#### 4.4.4 The First Interest Statement

Now that we have a database populated with some artefacts, we can now use our object relational mapper to pull information out. Our first interest statement will be the simplest<sup>4</sup>:

I am interested in everything

In SQL syntax, this statement is equivalent to:

```
select * from artefacts
```

However, using ActiveRecord, we can also express this interest statement with Ruby code:

```
Artefact.find(:all)
```

We can even verify that the right results are being pulled out from the database with a simple console<sup>5</sup> session:

```
>> Artefact.find(:all).map {|artefact| artefact.shape}
=> ["box", "cylinder", "floor", "sphere"]
```

As can be seen, a simple query pulling out all the artefact shapes returns the shapes of the four artefacts that we loaded into the database with our yaml file. Inspecting one of the artefacts more closely, we can see that all the attributes are intact too:

```
>> print Artefact.find_by_shape("cylinder").to_yaml
--- !ruby/object:Artefact
attributes:
  radius: "1.0"
  x_coord: "10.0"
  y_coord: "5.0"
  id: "2"
  shape: cylinder
  colour: turquoise
  length:
  transparency: "0.5"
  height: "10.0"
  z_coord: "0.0"
  width:
```

---

<sup>4</sup>Ignoring the option of being interested in nothing at all.

<sup>5</sup>The Rails console is a standard Ruby interactive shell with all the necessary classes and objects pre-loaded into it (such as our Artefact class).

One thing that might look odd is that all the numerical data looks to be represented as strings. This is because ActiveRecord talks to the database using plain strings. However, in practice, this is not actually an issue. When the attributes are accessed using the standard accessors, ActiveRecord is able to translate the string to the correct class:

```
>> Artefact.find_by_shape('cylinder').x_coord.class
=> Float
```

Also, the virtual environment viewer, introduced in the following Section, deals entirely with strings.

#### 4.4.5 Viewing the Virtual Environment

With a working implementation of the core parts of a virtual environment necessary to build the interest management framework upon, it was necessary to build a tool allowing the visualisation of the artefacts within the world. This was achieved using a VRML browser (FreeWRL). VRML browsers typically only render static files which contain all the necessary information for the particular world/environment they are representing. In order to facilitate arbitrary addition and removal of VRML nodes, some VRML browsers offer an Extended Application Interface (EAI). The EAI allows the VRML browser to act like a server listening on a specific port. Communicating over this port a client that can control the VRML browser, i.e. sending commands to add and remove VRML nodes.

An abstraction layer was created which hid the unnecessary VRML syntax and the complexity of communicating through the EAI interface. This abstraction layer took the form of a standard UNIX command line interface, and provides the following options:

```
Welcome to the EAIShell. For help, just type help...

$>help

The following commands are available:

add                shape options
help shape         (for a list of available shapes)
help colours       (for a list of available colours)
help transparency  (for information about transparency)
hide              object_id
show              object_id
delete            object_id
set_transparency  object_id transparency_value
increment         coordinate object_id value
                  (where coordinate = x_coord, y_coord or z_coord)

help
```

```
quit or exit
$>
```

A Ruby wrapping library was built using this command line interface, which presented the VRML browser with a Viewer class which supports the same options as the EAIShell. The architecture of the viewing system is presented in Figure 4.1.

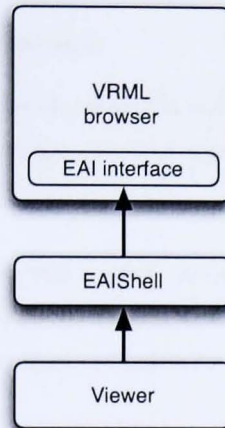


Figure 4.1: The Architecture of the Viewing System

`Viewer\#add`<sup>6</sup>, the add method that the Viewer class provides, takes a standard Ruby hash as its parameter. This hash represents all of the attributes of the particular artefact that you wish to add to the VRML browser. An example of this is as follows:

```
>> viewer = Viewer.new
>> viewer.add ( {:shape => :cube, :id => 1, :height => 10, :x_coord => 2, :y_coord => 4,
:z_coord => 6, :colour => "red", :transparency => 0.5 })
```

This hash is identical to the attributes hash that is included within each ActiveRecord Artefact object. This allows us to run the following simple Ruby code, to generate the world shown in Figure 4.2

```
>> viewer = Viewer.new
>> Artefact.find(:all).each {|artefact| viewer.add(artefact.attributes)}
```

<sup>6</sup>In standard Ruby documentation the # symbol is used to separate a class or module name from the method name

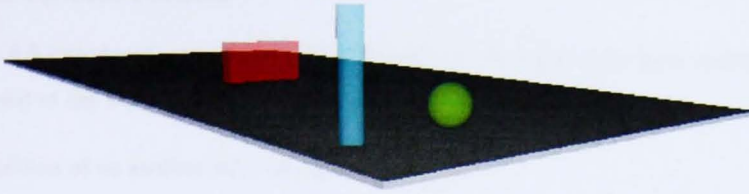


Figure 4.2: A Simple World Consisting of a Cuboid, Cylinder, Sphere, and Floor

#### 4.4.6 A Client Server Architecture

With the visualisation system in place, the next stage was to introduce a simple client server architecture into the implementation, as illustrated in Figure 4.3. This would represent the following logical separation of concerns:

**4.4.6.0.1 Client** The client interacts with a viewer and a server. It receives updates for the viewer from the server, and also registers its interests with the server (currently an interest in everything, as discussed in Section 4.4.4, but expanded upon in Chapter 5.)

**4.4.6.0.2 Server** The server interacts with a client and a database. It fetches the relevant artefacts from the database based on the client's interest, and sends them to the client. It also provides the ability for the client to register updates in interest.

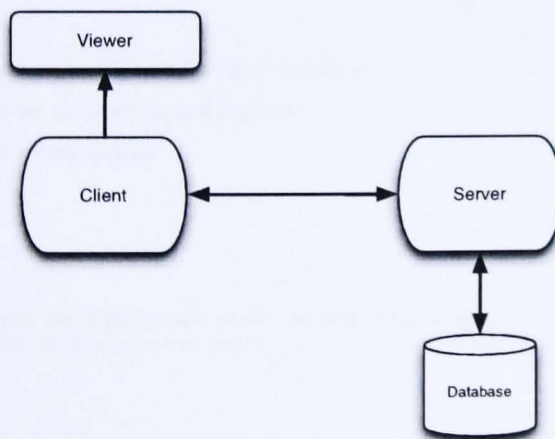


Figure 4.3: A Client Server Architecture

#### 4.4.7 An Update Format

As Section 4.4.6 introduced, one of the major concerns of the server is to send updates to the client. Updates consist of the following:

- The addition of an artefact into the client's view,
- The removal of an artefact from the client's view,
- The update of one or more attributes belonging to one or more artefacts within the client's view.

In order to be able to calculate which updates to send, the server needs to have a copy of what the client is currently aware of. When the server is sent the refresh method, it pulls out the latest set of artefacts from the database based on the interests. This new set can be compared against the stored set, and the differences sent as updates to the client. The algorithm is as follows:

```

for each interesting artefact
  if the client is not aware of the artefact:
    generate add command for the artefact
  end

  if the client is aware of the artefact, but the attributes are different:
    generate update commands for the artefact
  end
end

for each artefact in the copy of the artefacts the client is aware of
  if the artefact is not in the set of interesting artefacts:
    generate delete command for the artefact
  end
end

replace copy of artefacts that the client is aware of with the interesting artefacts

```



For example, let us look at the log of a simple session. On the first server refresh the following updates are sent to the client:

```
updates:
[
  {
    :command=>"add", :parameters=>{
      :y_coord=>20.0, :shape=>"box", :colour=>"red",
      :width=>2.0, :transparency=>0.5, :length=>10.0,
      :z_coord=>0.0, :height=>5.0, :id=>1, :x_coord=>50.0
    }
  },
  {
    :command=>"add", :parameters=>{
      :y_coord=>5.0, :shape=>"cylinder", :colour=>"turquoise",
      :radius=>1.0, :transparency=>0.5, :z_coord=>0.0,
      :height=>10.0, :id=>2, :x_coord=>15.0
    }
  },
  {
    :command=>"add", :parameters=>{
      :shape=>"floor", :width=>100.0, :transparency=>0.0,
      :length=>50.0, :id=>3
    }
  },
  {
    :command=>"add", :parameters=>{
      :y_coord=>15.0, :shape=>"sphere", :colour=>"green",
      :radius=>2.0, :transparency=>0.5, :z_coord=>0.0,
      :id=>4, :x_coord=>10.0
    }
  }
]
```

As the client currently has nothing visible, all the artefacts are added to the view. However if the server is refreshed again, the updates are as follows:

```
updates:
[]
```

Nothing has changed within the database since the last refresh, therefore there are no differences to send. If the x.coord of the sphere artefact is updated from 10.0 to 15.0, and the server refreshed, the updates are as follows:

updates:

```
{{:command=>"move", :parameters=>{:by=>5.0, :axis=>"x", :id=>4}}}
```

Only the differences are sent to the client. If cylinder artefact is removed from the database entirely, and the server refreshed again, the updates are as follows:

updates

```
{{:command=>"delete", :parameters=>{:id=>2}}}
```

The server is therefore correctly detecting any changes between the current set of interesting artefacts, and the set of artefacts the client has, and sending those changes across using the minimal amount of information.

We now have a fully working model of a the visual elements of a virtual environment<sup>7</sup>. The server is capable of sending updates to the client based on an interest in everything. The client starts the simulation with a knowledge of nothing, and everything is sent using the network which logs all throughput. This approach is similar to that taken by the Cyberwalk system[89]. We now have the potential to deal with a wide variety of heterogeneous clients by having close control of what is sent along the wire. This close control will be provided by the concept of interest which is the subject of the next chapter.

---

<sup>7</sup>It is important to note that this model ignores interactive elements such as navigation, and sending any information other than interests from the client to the server.

## Chapter 5

# Interest Statements

Chapters 3 and 4 considered the concept of dynamic interest management, and implementation of a basic virtual environment respectively. Given the implementation of a conceptual model representing a virtual environment (Section 3.1), it is necessary to consider how we might map on the concept of interests as defined in Section 3.2. Section 3.3 illustrated how this model is capable of representing many of the interest management techniques currently found in the literature (categorised in Section 2.2.3). This chapter will build upon this work by introducing an implementation of a language for interest statements.

Section 5.1 will describe the concepts that a language for interest statements will need to be able to express. Section 5.2 will revisit the examples introduced in Section 3.3, and Section 5.3 will illustrate how SQL is expressive enough to represent these examples. Section 5.4 will describe how to use SQL to combine the concerns described in Section 3.4.2, and finally Section 5.5 will discuss some of SQL's limitations as a language to represent interest statements. These limitations will be the main focus of Chapter 6.

### 5.1 Interesting Concepts

In order to express interest, it is necessary to consider the potential subjects of that interest: the concepts we wish to reason about. In essence we are looking to generate subsets of artefacts from the set of all artefacts where that subset is interesting. Therefore, implicitly, the interesting concepts are artefacts. However, in order to express which artefacts we are interested in, it is necessary to have some factors with which we can distinguish and reason with. Section 3.2.2 introduced the concept of interest conditions and defined them as *"a test which will indicate whether or not an artefact ( $x$ ) is interesting to us"*. The factors we need in order to define these tests are attributes, virtual attributes, relative artefacts and relative virtual artefacts. These will be discussed in the following sections.

### 5.1.1 Attributes

Attributes are the items of data which constitute an artefact. We can use the presence, absence or value of these to reason about different sets of artefacts. For example, we might make the following statement of interest:

*I am interested in all artefacts that are red*

This statement would be a comparison to a colour attribute, checking that the colour was red. All artefacts that have a colour attribute with the value red would be a member of the interesting set.

### 5.1.2 Virtual Attributes

Virtual attributes are attributes associated with an artefact, which are not necessary data for that artefact to exist. Essentially, virtual attributes are artefact metadata. For example, our virtual environment may contain artefacts representing people. Given the presence of a virtual attribute mood, we can make the following statement of interest:

*I am interested in all happy people*

This statement would add all people that have a mood attribute with the value happy to the interesting set.

### 5.1.3 Relative Artefacts

Attributes are strongly coupled to artefacts. Each attribute belongs to only one artefact and it represents information that is essential for the representation and existence of that artefact in the virtual environment. Although using artefact attributes to create interest statements provides us with a powerful and fine grained language to reason about interest, it is not enough to reasonably<sup>1</sup> represent many types of interest statement.

In the previous two examples of interest statements we injected values into the statement. In the first example we injected the value red, and in the second we injected the value happy. These were values we had to know before we could make the statement. Sometimes, however, we might not know these attribute values beforehand. For example, we might want to make the following statement:

*I am interested in all artefacts that are the same colour as artefact A*

Clearly, for this statement to make sense, we need to know which artefact *A* is referring to. This artefact is a relative artefact; it is the artefact which our statement requires in order to be complete. If we are to change the artefact, we potentially change the resulting interesting set.

<sup>1</sup>It would be possible to overload the interest statement to such an extent that all types of interest statement could be represented. However, this would require that any additional information other than artefact attributes be included within the interest statement itself.

### 5.1.4 Relative Virtual Artefacts

Relative virtual artefacts are a particular example of relative artefacts as defined in Section 5.1.3 in that they are also virtual artefacts. A virtual artefact is a non-essential artefact. Similar to the concept of a virtual attribute being artefact metadata, virtual artefacts are virtual environment metadata. A virtual artefact could be used to describe many different virtual concepts. For example they could be used to represent an aura such as one defined in Greenhalgh's spatial model of interaction[44]. In this model an aura is a virtual artefact defining an area, that moves with the associated artefact. An aura does not 'exist', but it does 'virtually exist' within the virtual environment, and is therefore particularly suited to be represented with a virtual artefact. Similarly, virtual artefacts could also be used to define locales.

Examples of this additional information are the spatial and zonal geometrical representation used for locales and the focus and nimbus areas. A solution to this is to move this information from the interest statement into the virtual environment. The information can be represented with virtual artefacts. A virtual artefact is an artefact that cannot be directly interacted with. It is essentially invisible information. Although the distinction between the terms *virtual artefact* and *artefact* are nonsensical in the context of a virtual environment (where everything is essentially virtual), it is useful to conceptually separate them. The separation is only conceptual because in terms of implementation, a virtual artefact is essentially just an artefact, but it is in their usage that they differ. Virtual artefacts are, in essence, information about information, and therefore the phrase *virtual artefact* is just another term for metadata.

## 5.2 Example Statements

This section will explore some examples of the sort of statements that the language needs to be able to represent. The motivation of this section is to revisit the examples introduced in Section 3.3 within the context of the interest concepts as defined in Section 5.1.

### 5.2.1 Categories

As an example of categories, Section 3.3.4 described the interesting set consisting of all red artefacts. This type of statement is represented in terms of the relationship between the statement's given values and the artefact's current attribute values:

*I am interested in all artefacts that are red*

### 5.2.2 Locales

Locales can be seen to be either relative artefacts or relative virtual artefacts. An example of a relative artefact representing a locale is a building, and an example of a relative virtual artefact is an area or zone

represented by a virtual artefact. Such zones appear in many different explicit guises in the real world. Examples include countries, building plots, and even different sections of a sports area such as a football pitch. If an artefact (virtual or not) is to be used as a locale, it needs to have attributes that describe its spatial relationship between other artefacts (such as co-ordinates, and bounding boxes). Artefacts representing locales should also support the standard spatial set relationships such as: *within*, *contains*, *overlaps*, and *touches*.

An example of an interest statement using locales (using a relative artefact) is:

*I am interested in all artefacts within the football pitch*

Similarly, an example of an interest statement using locales (using a virtual relative artefact) is:

*I am interested in all artefacts within my viewing frustum*

### 5.2.3 Relative Locales

In terms of the interest concepts as defined in Section 5.1, relative locales can be seen to be the same as standard locales. This is because locales are represented by artefacts or virtual artefacts, and any artefact (virtual or not) represented in an interest statement is a relative artefact. Where there is a relationship between two or more artefacts, such as the relationship described in Section 3.3.2, this relationship should be part of the declaration of the relative artefact. Therefore, if we can refer to the relative artefact *a*, we can also refer to an artefact (*b*) related to *a*. The interest statement does not contain any information about relationships between artefacts.

### 5.2.4 Interacting Locales

Section 3.3.3 described an example of interacting locales as follows

*"When an artefact A's aura collides with artefact B's aura, artefact A can be said to be aware of artefact B."*

In terms of the interest concepts as defined in Section 5.1, concepts such as the one above can be represented by introducing two relative artefacts into the interest statement:

*I am interested in all artefacts whose aura overlaps my aura*

### 5.2.5 Combinations

We might wish to be able to create statements that arbitrarily combine the examples above. For example, consider the following interest statement:

*I am interested in all artefacts that are red and also within my viewing frustum*

All of these example statements will be revisited Sections 5.3 where they will be discussed within the context of an implementation.

## 5.3 Representing Interest Statements with SQL

In order to realise the examples highlighted in Section 5.2 it is necessary to consider mechanisms with which to represent interest statements. Given that the virtual environment's data is stored using a database (see Section 4.3.1.5), and that the model of interest management introduced in Chapter 3 was in terms of set theory, SQL seems a very obvious tool with which to represent our interests. This section will explore the use of SQL in detail, concluding that SQL is expressive enough to represent the examples in Section 5.2.

This section will explore SQL's ability to represent the examples presented in Section 3.3 in order to demonstrate the expressiveness of SQL for representing interest statements.

### 5.3.1 Categories

Section 5.2.1 introduced the following as an example of a category based interest statement:

*I am interested in all artefacts that are red*

Section 3.3.4 introduced the following general case for representing category based interest statements:

$$\mathcal{I} = \{x \in \mathcal{U} : x \in \text{CATEGORY}\} \quad (5.1)$$

The following scopes the general case to our specific example of red artefacts:

$$\mathcal{I} = \{x \in \mathcal{U} : x \in \text{RED\_ARTEFACTS}\} \quad (5.2)$$

where  $\text{RED\_ARTEFACTS}(x)$ :  $x$  is red

Which can be simplified to:

$$\mathcal{I} = \{x \in \mathcal{U} : x \text{ is red}\} \quad (5.3)$$

This is the statement that we need to express using SQL. In order to achieve this, it is necessary to evaluate each concept in the set builder in terms of SQL. Table 5.1 explores these individual concepts in turn.

Therefore, the corresponding SQL statement for the category based example interest statement is as follows:

Table 5.1: Deconstructing and Evaluating a Category Based Set Builder

Set Builder Concept	Description	Equivalent SQL Concept
$x \in$	Any artefact within	<code>select * from</code>
$\mathcal{U}$	the universal set, the set of all artefacts	<code>Artefacts</code>
:	such that the following is true:	<code>where</code>
$x \text{ is red}$	the artefact is red	<code>colour = 'red'</code>

```
select * from Artefacts where colour = 'red'
```

Notice that we can use the standard SQL conditional operators to compare attributes (colour) against given values ('red'):

Table 5.2: SQL Conditional Operators

Conditional Operator	Description
<code>=</code>	equal to
<code>!=</code>	not equal to
<code>&lt;</code>	less than
<code>&gt;</code>	greater than
<code>&lt;=</code>	less than or equal to
<code>&gt;=</code>	greater than or equal to

### 5.3.2 Locales

Section 5.2.2 introduced the following as an example of a locale based interest statement:

*I am interested in all artefacts within the football pitch*

Section 3.3.1 introduced the following general case for locale based interest statement:

$$\mathcal{I} = \{x \in \mathcal{U} : x \text{ is within } A'\} \quad (5.4)$$

The following scopes the general case to our specific example of red artefacts:

$$\mathcal{I} = \{x \in \mathcal{U} : x \text{ is within the football pitch}\} \quad (5.5)$$

In order to convert this set builder to an interest statement, it is necessary to define the relationship *within*, particularly in terms of football pitches. One way of representing the area of a football pitch is with a two dimensional rectangle. Consider the football pitch represented in Figure 5.1. This football pitch has the following properties which are interesting in this context: a width, a length and a pair of coordinates which define the centre of the pitch.



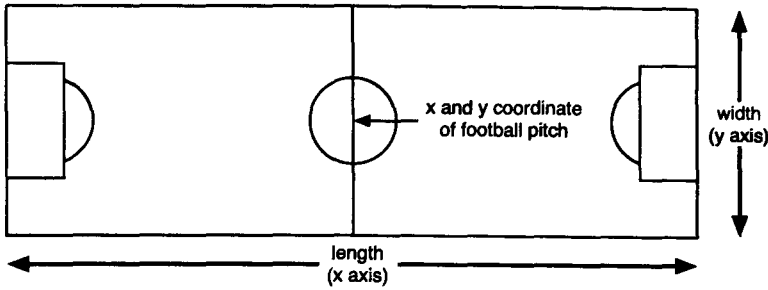


Figure 5.1: A Simple Representation of a Football Pitch

A simple algorithm for determining whether a given artefact's  $x$  and  $y$  coordinates fall within the area of the football pitch is illustrated in Figure 5.2. Essentially it is determining whether the artefact's coordinates are on the pitch side of each of the four pitch boundaries. If this is the case for all the boundaries, then the artefact is considered to be within the football pitch. The use of range predicates over multiple artefact attributes for representing spatial regions of interest is an approach also used by Bharambe et al.[15].

The following is a partial SQL statement which represents the *within* algorithm:

```
x_coord >= football_pitch.x_coord - (football_pitch.length / 2) and
x_coord <= football_pitch.x_coord + (football_pitch.length / 2) and
y_coord >= football_pitch.y_coord - (football_pitch.width / 2) and
y_coord <= football_pitch.y_coord + (football_pitch.width / 2)
```

However, in order to convert this to a full SQL statement it is necessary to introduce the concept of relative artefacts. In the snippet above, `football_pitch` represents a relative artefact - an artefact that the interest statement is relative to. One way of introducing relative artefacts into the SQL statement is through a self join.

```
select a.* from Artefacts a, Artefacts football_pitch where
a.x_coord >= football_pitch.x_coord - (football_pitch.length / 2) and
a.x_coord <= football_pitch.x_coord + (football_pitch.length / 2) and
a.y_coord >= football_pitch.y_coord - (football_pitch.width / 2) and
a.y_coord <= football_pitch.y_coord + (football_pitch.width / 2) and
football_pitch.name = 'football pitch' and
category = 'pitch'
```

Here we are defining the relative artefact `football_pitch` by specifying its name and category (which in this case should be attribute columns which have the properties of a composite primary key).

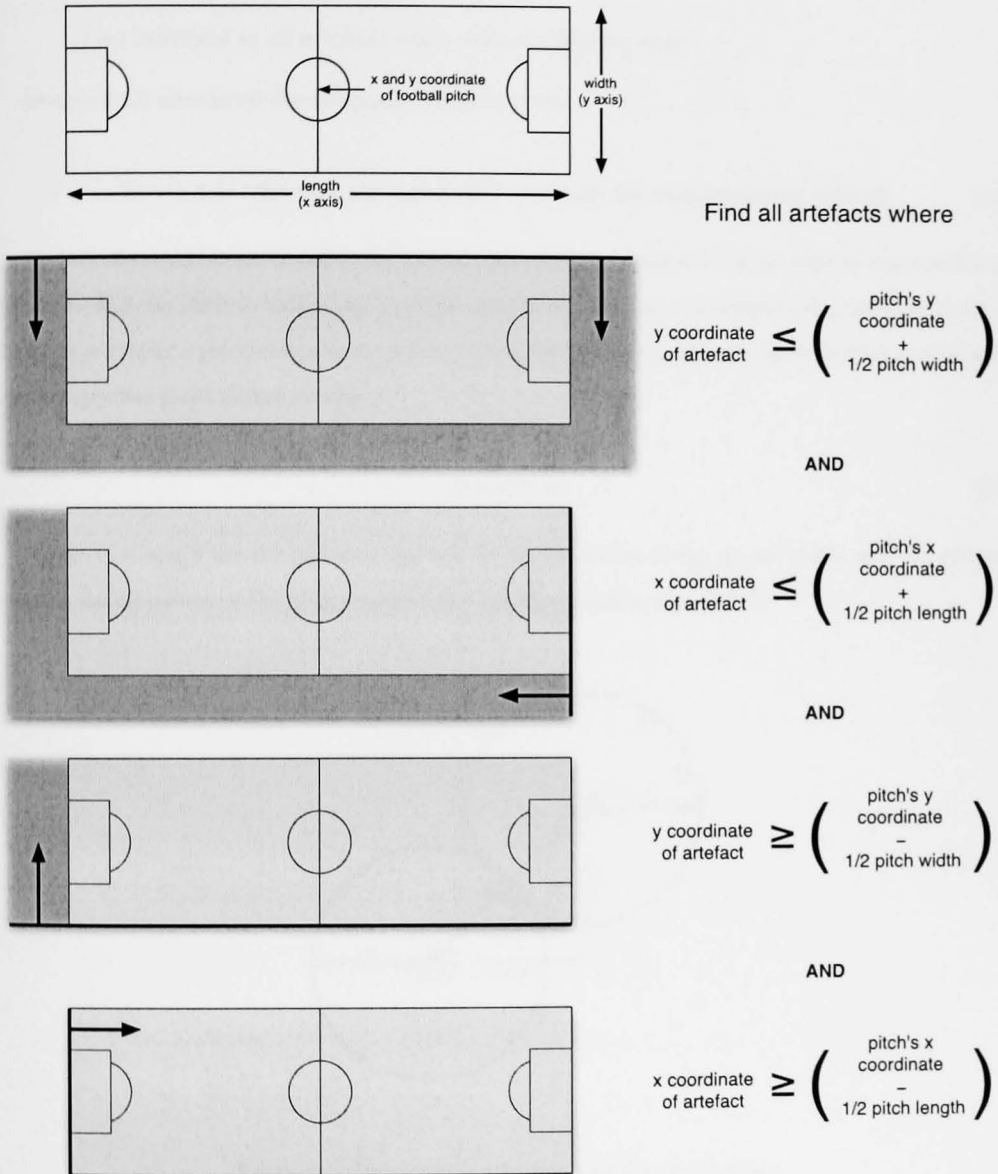


Figure 5.2: Determining whether a Given Artefact's x and y Coordinates Fall Within the Area of a Football Pitch

### 5.3.3 Interacting Locomes

Section 5.2.4 introduced the following as an example of an interest statement in terms of interacting locales:

*I am interested in all artefacts whose aura overlaps my aura*

Section 3.3.3 introduced the corresponding set builder:

$$\mathcal{I} = \{x \in \mathcal{U} : \text{the aura associated with } a \text{ overlaps the aura associated with } x\} \quad (5.6)$$

As with the requirement to define the *within* relationship in Section 5.3.2, in order to convert this set builder to SQL we need to define the *overlaps* relationship. In order to simplify the example, consider the relative artefacts representing each aura to be circular<sup>2</sup>. We need to define an algorithm to determine whether any two given circles overlap.

$$r + r' > d \quad (5.7)$$

Where  $r$ ,  $r'$  and  $d$  are the radius of the first circle, the radius of the second circle, and the distance between the midpoints of the circles respectively (as illustrated in Figure 5.3).

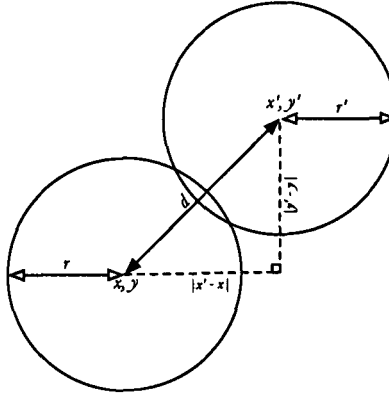


Figure 5.3: Determining whether Two Circles Overlap

Notice that distance  $d$  is the hypotenuse of the dotted triangle, the lengths of which other sides are  $|x' - x|$  and  $|y' - y|$ . We can therefore use Pythagorus' theorem to calculate  $d$  as follows:

$$d = \sqrt{|x' - x|^2 + |y' - y|^2} \quad (5.8)$$

<sup>2</sup>This approach is also taken by the Cyberwalk system[89]

Therefore, to determine whether two circles overlap, the following condition should be true:

$$r + r' > \sqrt{|x' - x|^2 + |y' - y|^2} \quad (5.9)$$

In order to create the SQL statement which represents this example, we need to consider the additional constraints on the artefacts. We are looking for all artefacts that have the same name as a aura that overlaps the aura corresponding to 'my artefact'. The following SQL statement represents this example<sup>3</sup>:

```
select c.* from Artefacts a, Artefacts b, Artefacts c where
    a.name = 'my artefact' and
    a.category = 'focus' and
    b.category = 'nimbus' and
    c.name = b.name and
    c.virtual = false and
    a.radius + b.radius > sqrt(pow((b.x_coord - a.x_coord), 2) + pow((b.y_coord - a.y_coord), 2))
```

### 5.3.4 Combinations

Section 5.2.5 introduced the following as an example of a combinational interest statement:

*I am interested in all artefacts that are red and also within my viewing frustum*

Let us treat each part of this statement individually before combining them together. The two parts are:

*I am interested in all artefacts that are red*

and

*I am interested in all artefacts that are within my viewing frustum*

The first part is simply a category based interest statement as discussed in Section 5.3.1. The SQL representation of which was:

```
select * from Artefacts where colour = 'red'
```

The second part is a locale based interest statement as discussed in Section 5.3.2. However, Section 5.3.2 described a rectangle as the shape of the locale. In the context of viewing frustums, this might not be an appropriate shape. Instead, let us consider that a viewing area is circular, centred around the associated artefact. We need to define the relationship *within* with respect to a circle. For a given

---

<sup>3</sup>The attributes name, category and virtual are simply example attributes, and are used and introduced further in Section 7.1.1.

artefact to be within a given circle, its coordinates must be within the area of the circle. This means that the distance between the centre of the artefact and the centre of the circle must be less than the circle's radius (as illustrated in Figure 5.4):

$$r > d \quad (5.10)$$

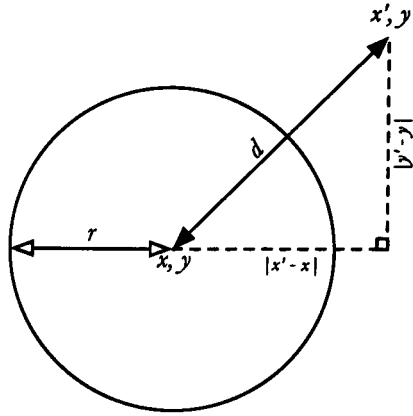


Figure 5.4: Determining whether a Given Point is Within a Circle

The calculation of  $d$  is the same as described in Equation 5.8 in Section 5.3.3:

$$d = \sqrt{|x' - x|^2 + |y' - y|^2} \quad (5.11)$$

Therefore, to determine whether a coordinate is within a given circle, the following condition should be true:

$$r > \sqrt{|x' - x|^2 + |y' - y|^2} \quad (5.12)$$

The SQL for this part is therefore:

```
select b.* from Artefacts a, Artefacts b where
    a.name = 'my artefact' and
    a.category = 'frustum' and
    a.radius > sqrt(pow((b.x_coord - a.x_coord), 2) + pow((b.y_coord - a.y_coord), 2)) and
    b.virtual = false
```

Again, we are assuming that the frustum has the same name as the artefact it is associated with.

In order to combine these two parts we just need to utilise the SQL `and` and `or` keywords to join the interest statements. A combined version of the SQL interest conditions above is as follows:

```
select b.* from Artefacts a, Artefacts b where
    a.name = 'my artefact' and
    a.category = 'frustum' and
    a.radius > sqrt(pow((b.x_coord - a.x_coord), 2) + pow((b.y_coord - a.y_coord), 2)) and
    b.virtual = false and
    b.colour = 'red'
```

## 5.4 Combining Separate Concerns

Section 3.4.2 introduced the notion of the separation of concerns of the user and the simulation. It proposed the following concerns:

- User Positive Interests (see Section 3.4.2.1),
- User Negative Interests (see Section 3.4.2.2),
- Simulation Positive Interests (see Section 3.4.2.3),
- Simulation Negative Interests (see Section 3.4.2.4).

It then proposed the following method of combining these concerns:

$$\text{Interesting Artefacts} = ((\text{UPOS} - \text{UNEG}) \cup \text{SPOS}) - \text{SNEG} \quad (5.13)$$

Where:

Abbreviation	Full Name of Set
UPOS	The user's positive enforcements
UNEG	The user's negative enforcements
SPOS	The simulations's positive enforcements
SNEG	The simulations's negative enforcements

Within the context of SQL, each of these concerns can be represented with an SQL statement, and combined using the SQL operators as follows:

```
select * from Artefacts where (((id in (select id from Artefacts where UPOS) and
                                not id in (select id from Artefacts where UNEG)) or
                                id in (select id from Artefacts where SPOS)) and
                                not id in (select id from Artefacts where SNEG))
```

For example, the following set of combined interest statements will result with an interest in white or green artefacts only:

```
select * from Artefacts where (((id in (select id from Artefacts where (colour = 'white' or
                                                                    colour = 'red' or
                                                                    colour = 'blue')) and
not id in (select id from Artefacts where (colour = 'blue')))) or
id in (select id from Artefacts where (colour = 'green')))) and
not id in (select id from Artefacts where (colour = 'red')))
```

## 5.5 Limitations

Although, as shown in Section 5.3, SQL is expressive enough for our needs, it also has some limitations which might hinder its usage in this domain. This section will explore some of these limitations, particularly the issues of abstraction, readability, and succinctness.

### 5.5.1 Expressiveness

Section 3.2.2 introduced the concept of interest conditions. This chapter has only discussed simple conditions such as equality and mathematical inequalities such as greater than and less than. However, we may want to utilise much richer conditions such as visible, friend or threatening. Unfortunately, standard SQL is not capable of expressing this logic directly. This limitation has given rise to embedded procedural languages within SQL statements such as Oracle's Procedural Language/Structured Query Language (PL/SQL). This limitation is discussed further in Section 8.3.4.

### 5.5.2 Abstraction

Consider the following simple interest statement in SQL:

```
select * from Artefacts where colour = 'red'
```

This statement is fairly simple, and digestible. However, as we saw in Section 5.3.4, the statement gets more complex in proportion to the complexity of the the interest:

```
select b.* from Artefacts a, Artefacts b where
    a.name = 'my artefact' and
    a.category = 'frustum' and
    a.radius > sqrt(pow((b.x_coord - a.x_coord), 2) + pow((b.y_coord - a.y_coord), 2)) and
    b.virtual = false and
    b.colour = 'red'
```

Ideally we would like to keep the complexity of the statement we are currently writing to a minimum, yet still handle increasingly complex statements. SQL does not provide any such abstraction technique.

### 5.5.3 Readability

Consider the following SQL statement which determines which artefacts are within a given artefact's viewing frustum:

```
select b.* from Artefacts a, Artefacts b where
    a.name = 'my artefact' and
    a.category = 'frustum' and
    a.radius > sqrt(pow((b.x_coord - a.x_coord), 2) + pow((b.y_coord - a.y_coord), 2)) and
    b.virtual = false
```

Quickly scanning the above statement does not immediately reveal its intentions, particularly the line dealing with the geometry calculation of the distance between two coordinates.

### 5.5.4 Succinctness

Consider the following simple interest statement in SQL:

```
select * from Artefacts where colour = 'red'
```

As Section 5.5.2 described, this statement is fairly readable despite having a lot of words which are specific to the implementation (SQL) rather than the domain (interests). For example, in this statement the important concept is purely colour = 'red'. The lack of modularity, as described in Section 5.5.2, also means that in terms of the intention, the statement is not very succinct. Consider the differences between this valid SQL statement, and the following invalid statement which uses abstraction to define the within relationship:

```
select b.* from Artefacts a, Artefacts b where
    a.radius > sqrt(pow((b.x_coord - a.x_coord), 2) + pow((b.y_coord - a.y_coord), 2))

all artefacts where
    within_circle a
```

By increasing the succinctness in terms of the intention through modularity, the above invalid statement also becomes a lot more readable without losing the expressiveness. Chapter 6 will introduce Wish, a new language built upon SQL which aims to tackle the readability, succinctness and abstraction issues raised in this section.



## Chapter 6

# Wish: a DSL for Interest Statements

Chapter 4 introduced an implementation of a virtual environment that allowed interest to be represented using SQL, and Section 5.3 demonstrated that SQL is expressive enough to represent the interest types categorised in Section 2.2.4<sup>1</sup>. However, Section 5.5 described several limitations of using SQL to represent interest statements. SQL's limitations include a lack of support for abstraction/modularity mechanisms, and not being very readable or succinct. There is therefore the need for a language that is as expressive as SQL, provides abstraction mechanisms, and is also more suited to the domain of interests in terms of readability and succinctness. Wish aims to be such a language.

Section 6.2 describes the structure of Wish, and Section 6.3 describes how it was implemented. Wish is evaluated in Chapter 7.

## 6.1 The Structure of a DSL for Interest Statements

On inspection of the SQL interest statements described in Section 5.3 we can see that they consist of the following fundamental concepts: *interest conditions* or *relative interest conditions* connected with *logical operators* with optional *grouping*. Sections 6.1.2.1, 6.1.2.2 and 6.1.2.4 will explore these concepts in more detail. Two further concepts, not explicitly incorporated into the previous SQL interest statements, are needed to improve the readability and abstraction capabilities of a language representing interest statements. These two concepts are scoping and abstraction and will be discussed in Sections 6.1.2.6 and 6.1.2.5 respectively.

### 6.1.1 Domain Objectives

The domain objectives are driven by the successes and failings of the SQL implementation introduced in Chapter 5. A domain specific language needs to be expressive enough to represent all the examples described in Section 5.1.4, yet not suffer from the limitations discussed in Section 5.5. This section will discuss these objectives in greater detail.

---

<sup>1</sup>However, some of the predicates may best be calculated with auxiliary logic, such as visibility-based filtering.

### 6.1.1.1 Abstraction

A statement of interest can be formed with arbitrary amounts of complexity. However, complex statements can get very difficult to manage, thus increasing the chance of errors, both semantic and syntactic. The main problem with this situation is that the complexity of any linguistic statement is limited to the complexity that the author of the statement can handle.

One of the key techniques of dealing with complexity is through abstraction. By building many layers of abstractions, we are able to construct arbitrarily complex statements that only expose the complexity that is necessary in a particular context. One way of providing layers of abstraction is allowing the language to be built up from smaller modules, which in turn may be built up of smaller modules ad infinitum. These modules can be seen to be analogous to functions or classes.

### 6.1.1.2 Succinctness

When attempting to make a language more readable, it is possible to end up with something that is verbose. It is therefore important to find the correct balance between succinctness and readability. One method of doing this is to just remove semantic and syntactic elements that are not necessary in the current context. Consider the following SQL example describing an interest in all red artefacts:

```
select * from artefacts where (colour = 'red')
```

This is a complete SQL statement, and contains all the semantic and syntactic elements to be a correct SQL statement. However, in this context we are only considering interests. There are a few assumptions we can make. For example, consider the following assumptions:

- The universal set is always the set of all artefacts. Therefore, `from artefacts` is not necessary:

```
select * where (colour = 'red')
```

- We are always interested in all interesting things. Therefore, `select *` is not necessary:

```
where (colour = 'red')
```

- We are always describing interests. Therefore, `where` is not necessary:

```
(colour = 'red')
```

Through the process of making assumptions from the context of the domain, it is possible to remove elements of the language, and make the statement more readable.

### 6.1.1.3 Readability

One of the main aims for the readability for a new language is for it to be readable by the domain expert. The domain expert may not be a programmer, but someone designing the methods or techniques. Therefore, ideally, the language uses the vocabulary of the domain, and not necessarily the vocabulary of a general programming language.

For example, consider the following SQL snippet:

```
(colour = 'red')
```

This snippet, although fairly succinct<sup>2</sup> is perhaps not as readable as it could be. It still looks more like an SQL snippet, than a statement of interest. Consider the following possible methods of tackling this issue:

- The parenthesis aren't necessary in this statement. There are no operators to which their precedence can be altered.

```
colour = 'red'
```

- The quotations aren't necessary in this statement. The fact that the colour is represented by letters, means that it's possible to deduce that it is a string<sup>3</sup>:

```
colour = red
```

- It would be much more readable if it could be written within the language of the domain. In this case we are concerned with red artefacts, or all artefacts coloured red. Therefore the following would be more readable:

```
coloured red
```

Through the process of purging the statement from syntactic clutter, and using domain specific vocabulary, it is possible to make the statement more readable.

### 6.1.1.4 Expressiveness

In order to represent complex statements at all, the language needs to be sufficiently expressive. The requirement for expressiveness was captured in Chapter 3, and a test of which is the ability to represent the examples presented in Section 3.3.

---

<sup>2</sup>At least it is succinct when compared to the full SQL version

<sup>3</sup>More on this type of inference in Section 6.2.1.1.

## 6.1.2 Structural Concepts

This section evaluates the interest statements formalised in Chapter 3, and implemented in Chapter 5 in order to highlight the various structural concepts required of such a language. The following structural concepts are considered:

- Interest Conditions (Section 6.1.2.1),
- Relative Interest Conditions (Section 6.1.2.2),
- Logical Operators (Section 6.1.2.3),
- Grouping (Section 6.1.2.4),
- Abstraction (Section 6.1.2.5),
- Scoping (Section 6.1.2.6).

### 6.1.2.1 Interest Conditions

The fundamental component of the SQL interest statements introduced in Chapter 5 is a boolean expression. This is essentially a condition over a given attribute<sup>4</sup>:

*[attribute] [condition] [value]*

The SQL conditional operators were described in Table 5.2 and are reproduced here for convenience:

Conditional Operator	Description
=	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Example boolean conditions are as follows:

`colour = 'red'`

`age > 27`

`virtual != true`

---

<sup>4</sup>The square brackets in these descriptions do not denote optional tokens, they just emphasise token boundaries.

### 6.1.2.2 Relative Interest Conditions

Relative boolean conditions allow for the representation of statements such as

*I am interested in all artefacts that are the same colour as this artefact*

where *this* is a given artefact (as described in Section 5.1.3). These conditions appear as follows:

[*attribute*] [*condition*] [*a given artefact's attribute*]

Example relative boolean conditions are as follows (relative to artefact *x*):

`colour = x.colour`

`age > x.age`

`virtual != x.virtual`

### 6.1.2.3 Logical Operators

An interest statement can consist of one or more boolean conditions (or relative boolean conditions) joined together with logical combinational operators. Table 6.1 describes the four major logical combinational operators used for joining interest conditions.

Table 6.1: Logical Combinational Operators

Logic Symbol(s)	SQL Equivalent
$\vee$	or
$\wedge$	and
$\wedge \neg$	and not
$\vee \neg$	or not

An example of joining two interest conditions is:

[*interest condition*] [*logical operator*] [*interest condition*]

Examples of joining interest conditions are as follows:

`colour = 'red' or age > 27`

`colour = 'red' and age > 27 and not virtual = true`

Another logical operator that can be used is the `not` operator. This operator is not used to combine interest statements, but is used to switch the boolean value of a single interest statement.

[*not*] [*interest condition*]

An example of using the not operator on an interest statement is as follows:

```
not colour = 'red'
```

It is essentially equivalent to this statement:

```
colour != 'red'
```

However, the not operator becomes particularly useful with a language that provides grouping and abstraction as will be discussed in Sections 6.1.2.4 and 6.1.2.5 respectively.

#### 6.1.2.4 Grouping

Grouping allows a number of interest statements to be treated as a single interest statement by the logical operators. In SQL grouping is possible through the use of parenthesis. For example, the following interest conditions have been grouped together:

```
(colour = 'red' or age > 27)
```

The above statement is syntactically identical to the following statement:

```
colour = 'red' or age > 27
```

However, the presence of grouping allows logical operators to apply to a number of combined interest conditions simultaneously. The following are examples of logical operators being applied to groups of interest conditions:

```
not (colour = 'red' or age > 27)
```

```
(colour = 'red' or age > 27) or virtual != false
```

#### 6.1.2.5 Abstraction

Section 5.5.4 introduced an example of abstraction in order to provide an example of succinctness. It described how the following statement:

```
select b.* from Artefacts a, Artefacts b where
    a.radius > sqrt(pow((b.x_coord - a.x_coord), 2) + pow((b.y_coord - a.y_coord), 2))
```

The above statement might be abstracted to the following statement which looks similar to a standard programming language function call:

```
all artefacts where
    within_circle a
```

The important concept to note is that the implementation details are hidden away beneath a layer of abstraction. Notice that the above abstraction condition used a relative artefact as the parameter. The circle  $a$  needs to be defined somewhere for this condition to be meaningful. The usage of relative artefacts also removes the need for any self joins in the SQL select clause.

As described in Section 5.5.2, abstraction through modularity provides a way for the statement to express increasingly complex concepts whilst keeping the relative complexity and succinctness constant. If appropriate names are used for the abstractions, readability can also be improved.

### 6.1.2.6 Scoping

Scoping allows attributes to be matched against an entire set of values, rather than just one given value. It is represented by the SQL keyword `in` as seen in Section 5.4. An example of such an SQL statement is:

```
id in (select id from Artefacts where colour = 'red')
```

The above expression returns true if the `id` matches the `id` of a red artefact. If the attribute that we are scoping happens to be, or has the property of a primary key, the statement will behave the same as the following:

```
colour = 'red'
```

However, if the attribute does not have the property of a primary key, scoping becomes useful. Consider the following statement:

```
name in (select name from Artefacts where colour = 'red')
```

The above statement will match any artefact that has the same name as an artefact that is red. The semantics of this statement are not possible using standard interest conditions and logical operators as described above.<sup>5</sup> Scoping is particularly useful when used in conjunction with an abstraction mechanism as described in the next section.

## 6.2 Wish Structure and Syntax

Wish supports the following structural concepts: interest conditions, relative interest conditions, logical operators, grouping, scoping and abstraction. These concepts will be introduced in Section 6.1, and discussed in Sections 6.2.1, 6.2.2, 6.2.3, 6.2.4, 6.2.6, and 6.2.5 respectively.

---

<sup>5</sup>This is assuming the statement does not include any explicit joins in the select clause.

## 6.2.1 Interest Conditions

As introduced in Section 6.1.2.1, interest conditions are a fundamental component of an interest statement. In Wish an interest condition is bound to one line, and must conform to the following syntax (where the condition is one of the SQL logical operators presented in Table 6.1):

*[attribute] [condition] [value]*

Example Wish interest conditions are as follows:

`name = sam`

`age > 27`

`virtual != true`

### 6.2.1.1 Automagical Value Quoting

Consider the following interest conditions:

`name = sam`

`id = 3`

Notice the absence of any quotes, despite the value being a string and an integer consecutively. Wish will automatically quote most non-numerical and non-boolean values. The auto-quoting rules are as follows:

- Do not quote the value if any of the following are true:
  - it is numerical (e.g. 4, -4, 4.0, .5),
  - it contains a pair of expression tags<sup>6</sup> (e.g. `<x= x.value >`),
  - it is the word `true` or `false`,
  - it is surrounded by back-ticks (e.g. ``4 + 2``, ``sqrt(81)``),
  - it is already quoted.
- Surround the value with expression tags if all of the following are true:
  - it contains one or more periods within the value (e.g. `x.value`),
  - it does not start with a period (e.g. `not .rb`),
  - it does not end with a period (e.g. `not bye.`),

---

<sup>6</sup>The `<=` `>` tags will be covered in Section 6.2.2



- it does not contain any spaces (e.g. not hi. bye)
- it does not already contain any expression tags.
- Quote all other values

Therefore, to ensure that a value is quoted: quote it, and to ensure a value is not quoted: surround it with back-ticks. Examples of values before and after the automatic quoting mechanism are presented in Table 6.2.

Table 6.2: Value Quoting Examples

Before auto-quoting	After auto-quoting
4	4
'4'	'4'
four	'four'
4.0	4.0
cock-ver10	'cock-ver10'
1st	'1st'
true	true
false	false
truth	'truth'
x.value	<%= x.value %>
x.value.sub_value	<%= x.value.sub_value %>
'x.value.sub_value'	'x.value.sub_value'
<%= x.value %>	<%= x.value %>
<%= x.value %> + 1	<%= x.value %> + 1
end.	'end.'
.5	.5
.rb	' .rb'
2 + 4	'2 + 4'
the end is in sight	'the end is in sight'
.nearing.the.end	' .nearing.the.end'
nearly.at.the.end.	'nearly.at.the.end.'
Getting. Very. Close	'Getting. Very. Close'
.one.more.thing....	' .one.more.thing....'
`2 + 4`	2 + 4
`sqrt(81)`	sqrt(81)
The End.	'The End.'

### 6.2.1.2 Comments

Wish also supports comments. Any line that starts with a # is ignored. Consider the following Wish statement:

```
#this is a comment
```

Wish also ignores blank lines, i.e. lines containing only white-space characters (spaces and tabs).

## 6.2.2 Relative Interest Conditions

Section 6.1.2.2 introduced relative interest conditions as a fundamental interest statement structure. This essentially provides the ability to reference artefacts within an interest statement. For example, we might have a relative artefact `x` that we want to make our interest relative to. We might want to be able to state the following:

I am interested in all objects that are the same shape as `x`

Wish requires that `x` is defined in a file called `relative_artefacts.rb`. This file is essentially a series of variable declarations, where each variable declared needs to be assigned an ActiveRecord object representing an artefact<sup>7</sup>. Consider the following example `relative_artefacts.rb` file:

```
x = Artefact.find(1)
```

This file declares a variable (`x`) to which the artefacts with an id of 1 is assigned. It is important that the variable is assigned just a single artefact rather than a collection of artefacts. For example if there are multiple artefacts with the name 'sam', then `x` in the following statement is assigned a set of objects:

```
x = Artefact.find(:conditions => {:name => 'sam'})
```

Therefore, in the cases where the conditions do not exhibit the properties of a composite primary key, yielding only a single result, it is necessary to force ActiveRecord to only return one object. One way of achieving this is to select the first artefact only:

```
x = Artefact.find(:first, :conditions => {:name => 'sam'})
```

Using the defined relative artefact `x`, Wish can represent the relative interest condition as follows:

```
shape = x.shape
```

In the case where artefact `x` is a sphere, Wish will evaluate the above statement using Ruby to the following:

```
shape = sphere
```

It is also possible to evaluate more complicated expressions. For explicit evaluation, the expression tags `<%=` and `%>` are necessary. The tags surround the Ruby code which is to be evaluated. The tag and

Table 6.3: Evaluating Interest Condition Expressions

Before Evaluation	After Evaluation
shape = <%= x.shape %>	shape = sphere
age > <%= 20 + (5 * 2) - 3%>	age > 27
height < <%= x.height / 2%>	height < 5

its contents will be replaced by the appropriate value before the interest statement is executed. Consider the interest conditions and their corresponding evaluations presented in Table 6.3.

It is important to note that there are limitations for using these expressions to define relationships between artefacts. For example, although it would be easily possible to design an algorithm using this notation which defines a particular type of relationship, that particular algorithm could easily be extremely costly to execute, particularly if all users are attempting to execute the same algorithm simultaneously.

### 6.2.3 Logical Operators

Sections 6.2.1 and 6.2.2 described the syntax of individual interest conditions. We might also wish to use logical operators (such as those presented in Table 6.1) to create interest statements that contain multiple interest conditions. This section describes how this is achieved with Wish.

#### 6.2.3.1 not

Wish only supports the explicit logical operator `not` which must appear at the beginning of an interest condition line. For example consider the following interest statement:

```
not colour = 'red'
```

Therefore the complete syntax for an interest statement is:<sup>8</sup>:

```
(not) [attribute] [condition] [value]
```

#### 6.2.3.2 or

Wish does not explicitly support the `or` keyword. Given that each interest condition is represented by a separate line, Wish assumes that each new line with the same indentation as the previous line represents a combination of the two statements (current line and previous line) with an `or` operator.

For example, consider the following SQL snippet represents a combination of interest conditions:

```
colour = 'red' or age > 27
```

<sup>7</sup>This file is evaluated, and the relative artefacts fetched from the database, as part of the Wish compilation and the resulting values are used to generate the final SQL output.

<sup>8</sup>where the parentheses represent an optional token

This snippet can be expressed using Wish as follows:

```
colour = red
age > 27
```

### 6.2.3.3 and

In addition to the `or` keyword, Wish does not explicitly support the `and` keyword. Wish assumes that a new line with an increased indentation (two spaces<sup>9</sup>) represents a combination of the two statements with an `and` operator.

For example, consider the following SQL snippet which represents a combination of interest conditions:

```
colour = 'red' and age > 27
```

This snippet can be expressed using Wish as follows:

```
colour = red
  age > 27
```

### 6.2.3.4 and not, or not

It is possible to combine implicit `or`, implicit `and` and explicit `not` to create `or not` and `and not` representations for single interest statements.

For example, consider the following SQL snippet which uses `or not`:

```
colour = 'red' or not age > 27
```

This snippet can be expressed using Wish as follows:

```
colour = red
not age > 27
```

Also, consider the following SQL snippet which uses `and not`:

```
colour = 'red' and not age > 27
```

This snippet can be expressed using Wish as follows:

```
colour = red
  not age > 27
```

In order to apply the `not` keyword to more than one statement it is necessary to use the Wish grouping structures as described in Section 6.2.4.

<sup>9</sup>Wish does not support tabbing as an indentation token. All tabs are replaced with two spaces before the statement is parsed.

## 6.2.4 Grouping

Section 6.1.2.4 described how grouping allows a number of statements to be treated as one from the perspective of logical operators. Wish uses implicit and explicit grouping as described in Sections 6.2.4.1 and 6.2.4.2 respectively.

### 6.2.4.1 Implicit Grouping

Wish implicitly groups the following structures:

- individual conditions
- a block of conditions that are further indented from the current line

**6.2.4.1.1 Individual Conditions** For an example of implicit grouping involving individual statements, consider the following Wish statement:

```
not colour = red
```

Wish treats this line as one entire statement. The operators `not` and `=` only apply to this condition, and no others in the interest statement.

To see this behaviour explicitly, consider the following Wish statement:

```
not colour = red
age > 27
```

The above is equivalent to the following SQL snippet:

```
(not colour = red) or (age > 27)
```

**6.2.4.1.2 Indented Conditions** Wish implicitly groups all indented conditions. For an example of this, consider the following Wish statement:

```
colour = red
  age > 27
  name = sam
```

The above is equivalent to the following SQL snippet:

```
colour = red and (age > 27 or name = 'sam')
```

The grouping stops when the indentation returns to match the line where the grouping started. Consider the following statement:

```

colour = red
age > 27
name = sam
virtual = true

```

The above is equivalent to the following SQL snippet:

```
colour = red and (age > 27 or name = 'sam') or virtual = true
```

#### 6.2.4.2 Explicit Grouping

Wish supports explicit grouping through the use of grouping tokens. As with interest conditions, a grouping token must reside on its own line. The grouping tokens are as follows: `all` to start a group, and `not` to start a negated group. The grouping ends when a subsequent interest condition has the same indentation as the start token. For example, consider the following Wish statements:

```

all
  age > 27
  name = sam
virtual = false

```

The above is equivalent to the following SQL snippet:

```
(age > 27 or name = 'sam') or virtual = false
```

The structure of a negated group is identical:

```

not
  age > 27
  name = sam
virtual = false

```

The above is equivalent to the following SQL snippet:

```
(not (age > 27 or name = 'sam')) or virtual = false
```

Explicit groupings can contain explicit or implicit groupings. Consider the following statement which contains an implicit grouping within an explicit grouping:

```

not
  colour = red
  age > 27
  name = sam
virtual = false

```

The above is equivalent to the following SQL snippet:

```
(not (colour = red or age > 27 and (name = 'sam')))) or virtual = false
```

Wish also supports an optional group ending token: ---. When using both opening and closing grouping tokens, it is important to note that they both must appear at the same indentation. An example of an explicit grouping with an end token is as follows:

```
all
  colour = red
  age > 27
  name = sam
---
  virtual = false
```

The above is equivalent to the following SQL snippet:

```
(colour = red or age > 27 or name = 'sam') and virtual = false
```

## 6.2.5 Abstraction

Section 6.1.2.5 described abstraction as a useful tool for hiding complexity, and as a way of storing and re-using useful snippets of combined conditions, with an associated name. Storing such a combination is similar to defining a function or method in a programming language.

Wish offers an abstraction method through the creation and use of subwishes. Subwishes support the notion of implicit parameters, and can be nested within each other. They are discussed further in the following sections. Subwishes are essentially an implementation of derived sets as introduced in Section (3.2.4.1)

### 6.2.5.1 Subwishes

Wish supports abstraction through the concept of subwishes. Subwishes reside in their own files<sup>10</sup>, and the filename acts as the name of the subwish itself.

Any given Wish statement may contain one or more subwishes. Consider the following Wish statement:

```
#red_sphere.wish

#this subwish matches red spheres

colour = red

shape = sphere
```

---

<sup>10</sup>This is only a current implementation decision. The main concept is that subwishes are part of a hierarchy of Wish statements, and the subwishes can be used to provide a library of useful and frequently used partial Wish statement to use.

By storing it in a separate file with an appropriate name such as `red_sphere.wish`, it is possible to refer to it in the main interest using its name as follows:

```
red_sphere
```

Wish automatically replaces the abstraction with the content of the matching subwish's file, and wraps it up in an explicit grouping structure as follows:

```
all
  colour = red
  shape = sphere
...
```

This means that it is possible to use an abstraction in exactly the same fashion as a standard interest condition. For example, this Wish statement matches red spheres named sam:

```
red_sphere
  name = sam
```

Wish will automatically convert the above statement to the following:

```
all
  colour = red
  shape = sphere
...
  name = sam
```

### 6.2.5.2 Implicit Parameters

Section 5.3.2 introduced the following SQL snippet as a means of calculating whether a given artefact is within a football pitch:

```
x_coord >= football_pitch.x_coord - (football_pitch.length / 2) and
x_coord <= football_pitch.x_coord + (football_pitch.length / 2) and
y_coord >= football_pitch.y_coord - (football_pitch.width / 2) and
y_coord <= football_pitch.y_coord + (football_pitch.width / 2)
```

Assuming that we have an appropriately defined variable called `football_pitch` amongst the relative artefacts declarations, the Wish version is as follows:

```
x_coord >= <% football_pitch.x_coord - (football_pitch.length / 2) %>
x_coord <= <% football_pitch.x_coord + (football_pitch.length / 2) %>
```



```

y_coord >= <%= football_pitch.y_coord - (football_pitch.width / 2) %>
y_coord <= <%= football_pitch.y_coord + (football_pitch.width / 2) %>

```

Wish allows subwishes to contain parameters. Consider the following statement:

```
within football_pitch
```

Here we are passing the relative artefact `football_pitch` as a parameter to the subwish `within`. This subwish is stored in the file `within.wish` and would look as follows:

```

#within.wish

#this subwish determines whether a given artefact
#is within the boundaries of a rectangle
x_coord >= <%= |A|.x_coord - (|A|.length / 2) %>
x_coord <= <%= |A|.x_coord + (|A|.length / 2) %>
y_coord >= <%= |A|.y_coord - (|A|.width / 2) %>
y_coord <= <%= |A|.y_coord + (|A|.width / 2) %>

```

Notice how the above subwish refers to the relative artefact as `|A|`. When the Wish is parsed, all occurrences of `|A|` will be replaced with the first parameter of the subwish condition. This would result in the following:

```

all
x_coord >= <%= football_pitch.x_coord - (football_pitch.length / 2) %>
x_coord <= <%= football_pitch.x_coord + (football_pitch.length / 2) %>
y_coord >= <%= football_pitch.y_coord - (football_pitch.width / 2) %>
y_coord <= <%= football_pitch.y_coord + (football_pitch.width / 2) %>
...

```

If the subwish requires more than one parameter, they can be referred to as `|B|`, `|C|`, `|D|`, etc., in alphabetical order. For example, the following subwish takes two relative artefacts, and matches artefacts that are the same colour as the first, or the same shape as the second:

```

#colour_or_shape.wish

#this subwish takes two parameters

colour = |A|.colour
shape = |B|.shape

```

If it is stored in the file `colour_or_shape.wish`, then assuming the declaration of two relative artefacts `rel1` and `rel2` it could be called as follows:

```
colour_or_shape rel1 rel2
```

### 6.2.5.3 Nested subwishes

subwishes can be arbitrarily nested. It is possible to define a subwish which contains other subwishes, which in turn may contain other subwished etc.

For example consider the following subwish stored in a file called `red_artefact.wish`:

```
#red_artefact.wish
colour = red
```

This could be used by another subwish stored in a file called `red_sphere.wish` as follows:

```
#red_sphere.wish
red_artefact
shape = sphere
```

Which in turn could be used by the main Wish statement which would match all red spheres named sam:

```
red_sphere
name = sam
```

When using nested subwishes it is important to avoid creating closed cycles. For example, subwish A might refer to subwish B which in turn might refer back to subwish A. This would cause the Wish parser to enter an infinite loop, and is clearly undesirable.

### 6.2.6 Scoping

As described in Section 6.1.2.6, scoping allows attributes to be matched against arbitrary sets of values rather than just individual values. The Wish scoping mechanism uses subwishes to define a set of values. Consider the following subwish:

```
#blue_artefacts.wish
colour = blue
```

This subwish can be used to match blue artefacts as seen in Section 6.2.5.1. However, it can also represent the derived set of all blue artefacts (see Section 3.2.4.1). This set can be used in a scoping structure as follows:

```
name in blue_artefacts
```

This will match all artefacts that share a name with an artefact which happens to be blue (including the blue artefacts themselves). Scoping statements can also use parameters in an identical fashion to standard subwish calls. Consider the following scoping statement:

```
name in within football_pitch
```

The above statement will match all artefacts that share a name with an artefact which is within the relative artefact `football_pitch`.

### 6.2.7 Subwishes: A Myth

So far, this chapter has suggested an implicit difference between a standard Wish statement, and a subwish statement. However the difference was introduced purely for pedagogic purposes. Although there may be a conceptual difference in the context of a particular statement, there is no actual difference between Wishes and subwishes. Subwishes are just Wish statements that happen to be referred to by a different Wish statement.

Wish statements consist of a root Wish, which may (or may not) refer to other nested Wish statements, and a file describing any relative artefacts. Clearly, for a Wish statement to be used as a root Wish, it must not refer to any implicit parameters.

## 6.3 Design and Implementation

This section describes the implementation and design of the Wish language.

### 6.3.1 Agile Development

Wish was developed in an agile fashion with a focus on specifications, testing and modularity. Section 6.3.1.3 describes how Wish was built with a series of iterations, and Section 6.3.1.4 describes the process of one such iteration.

#### 6.3.1.1 Specifications

Wish was built using a behaviour driven development approach (see Section 4.3.2.2). This means that specifications of expected behaviour had to be written before the implementation of those behaviours could start. Developing Wish in this manner turned out to be invaluable, as the many constant changes to the implementation could instantly be verified.

#### 6.3.1.2 Modularity

Wish is a very modular system. This is due to two reasons: to allow the reuse of already available technologies such as YAML and Ruby erb, and also to facilitate the testing of the individual components. A modular architecture also lends itself particularly well to an iterative development process as a module can easily be the goal of a particular iteration.

### 6.3.1.3 Iterative Development

Wish was developed in an iterative fashion. An iteration is a small development cycle, with a small set of associated behaviours. The behaviours can be formalised using an executable specification language such as RSpec (discussed in Section 2.5.2). A typical iteration was as follows:

1. Decide on the objectives for the iteration
2. Define the behaviours of the objectives within the context of the system
3. Formalise the behaviours as a set of executable specifications
4. Append to the system implementation until the executable specifications pass

A fundamental principle of iterative development is that only the current iteration is defined and implemented. This allows the definition of the next iteration's objectives to be made based on the result of the previous iteration. This means that the overall development plan is agile and flexible, and able to deal with unanticipated situations. It also allows the development to focus on exactly what's necessary, and reduce the amount of unused implementation that was built on incorrect or outdated assumptions.

### 6.3.1.4 An Example Iteration

This section describes the process of a typical iteration. The objectives for this iteration are to implement the auto-quoting mechanism described in Section 6.2.1.1. The behaviours that the mechanism needed to exhibit were planned, and documented. They were essentially the behaviour documented in Table 6.2. These behaviours were converted to executable RSpec specifications, and are presented in Appendix A.3.

Once the specifications had been written, the implementation could commence. Appendix Implementation: A.1 presents the implementation which was the result of this particular iteration. When the implementation was completed the specifications were executed to validate that the implementation conformed to the specified behaviour. Appendix A.2, presents the output of the RSpec specification evaluation tool. Once the specifications all passed, it was important to run the whole suite of tests for the entire implementation so as to verify that the new implementation didn't affect any previous implementation. The next iteration was not allowed to start until all of the specifications passed.

## 6.3.2 Iterations

This section describes the various iterations that comprised the development of Wish. For each iteration, the goals and objectives are discussed.

### **6.3.2.1 Interest Conditions**

The first iteration of Wish developed the notion of a single interest condition. The implementation of such an interest condition was to be a simple SQL condition represented by a one element YAML list. This iteration also focussed on the infrastructure that allowed a single interest condition in it's own separate file to be used as an interest statement, and incorporated with the virtual environment implementation described in Chapter 4.

### **6.3.2.2 Explicit Logical Operators: not**

This iteration added specifications defining the behaviour of using the keyword `not` at the beginning of an interest condition to negate the result of that particular condition.

### **6.3.2.3 Implicit Logical Operators: or, and**

This iteration focussed on the ability to create interest statements that comprised of multiple interest conditions. It focussed on defining the behaviour of nested lists representing the `and` operator, and elements of the same list representing the `or` combinational operator as described in Sections 6.2.3.3 and 6.2.3.2 respectively.

### **6.3.2.4 Converting a YAML nested list to SQL**

This iteration focussed on the ability to convert the YAML nested lists that were the result of the previous iteration into a valid SQL where clause.

### **6.3.2.5 Expressions**

This iteration added specifications defining the behaviour of `erb` expressions and the relative artefacts file with respect to the Wish syntax.

### **6.3.2.6 Auto-quoting**

This iteration defined the auto-quoting behaviours described in Section 6.2.1.1. This particular iteration is described in greater detail in Section 6.3.1.4, and is the subject of Appendices A.3, A.1 and A.2.

### **6.3.2.7 Grouping**

This iteration focussed on specifying the behaviour of implicit and explicit grouping as discussed in Section 6.2.4.

### 6.3.2.8 Abstraction

This iteration focussed on the behaviour of an abstraction system building upon the grouping behaviour of the previous iteration. It specified that individual interest statements could be used as conditions within other statements by referring to the file name. A simple implicit parameter mechanism was also specified.

### 6.3.2.9 Scoping

This iteration defined specifications to support a simple scoping mechanism built upon the abstraction system. Scoping is discussed further in Section 6.2.6.

## 6.3.3 Architectural Components

Wish includes a number of existing technologies such as SQL, YAML and Ruby. Sections 6.3.3.1, 6.3.3.1, 6.3.3.3 describe the purpose of each of these existing technologies. Section 6.3.3.4 introduces the Wish layer, and finally Section 6.3.4.7 gives an overview of the various components that comprise Wish. Figure 6.1 illustrates these components, grouping them with the language that they are implemented in: YAML/SQL, Ruby and Wish respectively.

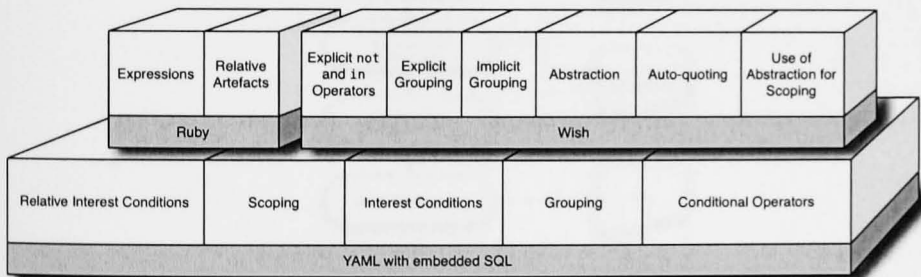


Figure 6.1: The Wish Components

### 6.3.3.1 SQL

SQL is the foundation language for Wish, and the final output language. Section 5.3 showed that SQL is suitably expressive as a language for representing interests. However, Section 5.5 described a number of limitations which motivated the creation of Wish. Figure 6.2 illustrates the overall process of compiling a Wish statement into an equivalent SQL statement.

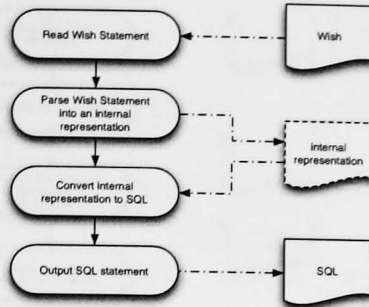


Figure 6.2: Converting a Wish Statement to SQL

### 6.3.3.2 YAML

Using YAML to define the structure of Wish allows the use of the YAML parser to create a data structure. Wish is essentially an arbitrarily nested list of interest conditions, and as discussed in Section 2.5.1, YAML allows this hierarchical data structure to be represented and parsed. Figure 6.3 illustrates the role of YAML as a format for representing an interest statement structure which can be easily parsed, and converted to SQL.

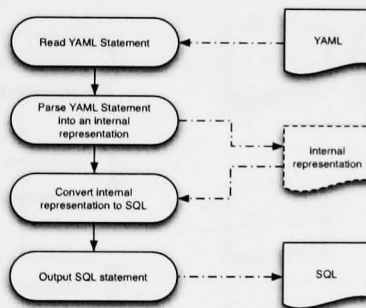


Figure 6.3: Parsing a YAML Statement and Converting it to SQL

### 6.3.3.3 Ruby

Wish uses Ruby to define relative artefacts and to evaluate expressions. Due to its interpretative nature, Ruby can evaluate arbitrary blocks of code embedded within text files. This is facilitated by the erb library which is discussed in Section 2.4.1. Ruby also allows a set of statements to be stored in a Binding object, and passed as a context to block, string eval or erb evaluations. This allows the definitions of relative artefacts to be stored in a separate file, and used as a context when evaluating the embedded erb statements. Figure 6.4 illustrates the role of Ruby erb expressions in the Wish parsing algorithm.

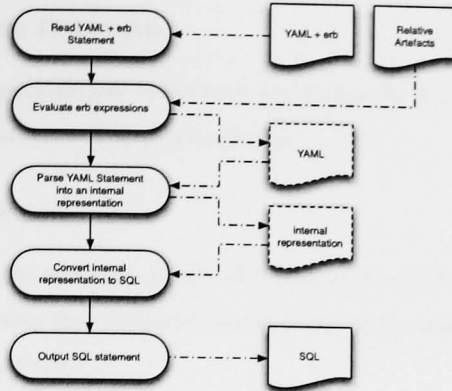


Figure 6.4: Parsing a YAML + erb Statement and Converting it to SQL

#### 6.3.3.4 Wish

Wish leverages the power of SQL, YAML and Ruby, and focusses on adding abstraction, grouping and simple scoping mechanisms whilst attempting to be both readable and succinct. The simple Wish parser essentially replaces Wish syntax with SQL conditions and stores the result in YAML format. Figure 6.5 illustrates the role of the Wish syntax in the Wish parsing algorithm.

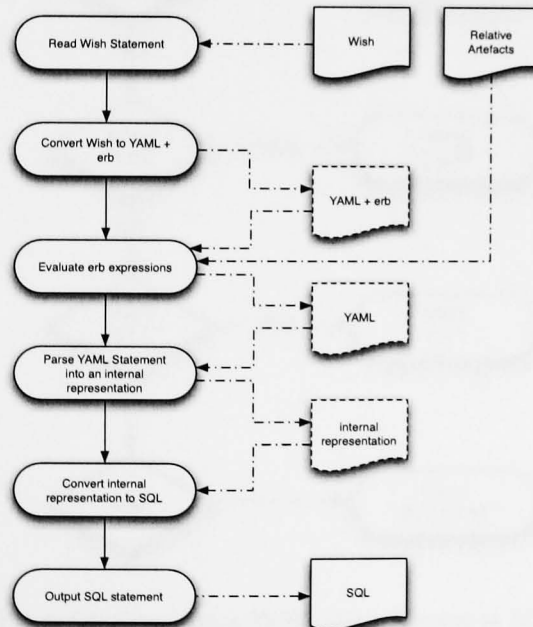


Figure 6.5: Parsing a Wish Statement and Converting it to SQL



### 6.3.4 Implementation Overview

This section describes the algorithms and methods used to implement Wish. Finally, Section 6.3.4.7 gives an overview of the components that Wish is comprised of.

#### 6.3.4.1 YAML to SQL Parser

The YAML to SQL parser has two main phases of operation. First, the YAML file is parsed by the Ruby YAML library into a Ruby data structure. Secondly, the Ruby data structure is manipulated and compiled into a valid SQL statement.

The compilation algorithm operates on a nested list, and is recursive in order to deal with the nested properties of the data structure. The base case for the algorithm is when the structure is a single element. Figure 6.6 describes this base case which is essentially a standard case statement.

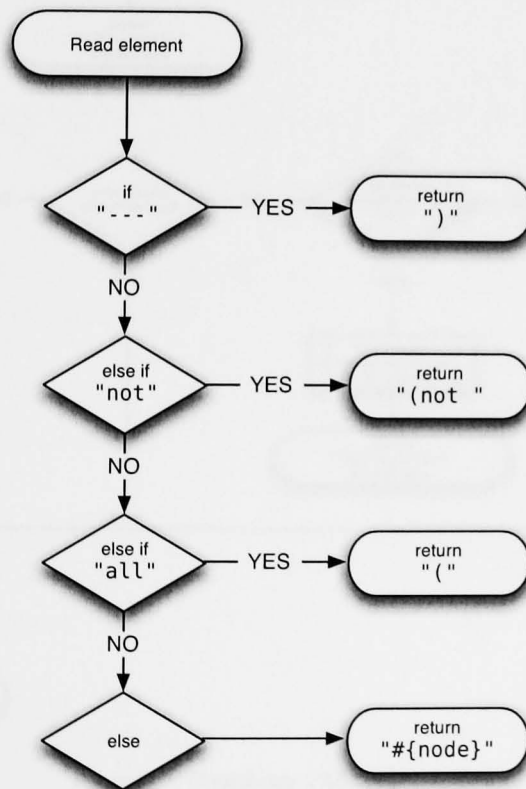


Figure 6.6: Converting a YAML String Element to SQL

If the algorithm is passed a list to compile, it iterates through each element and joins the result of a recursive call to the algorithm passing the current algorithm together with either an or or and keyword. The keyword to use depends on whether the current element is a single element, or a nested list. Figure

6.7 details the process of combining a list of elements, and Figure 6.8 describes the entire process of compiling a Wish statement to SQL.

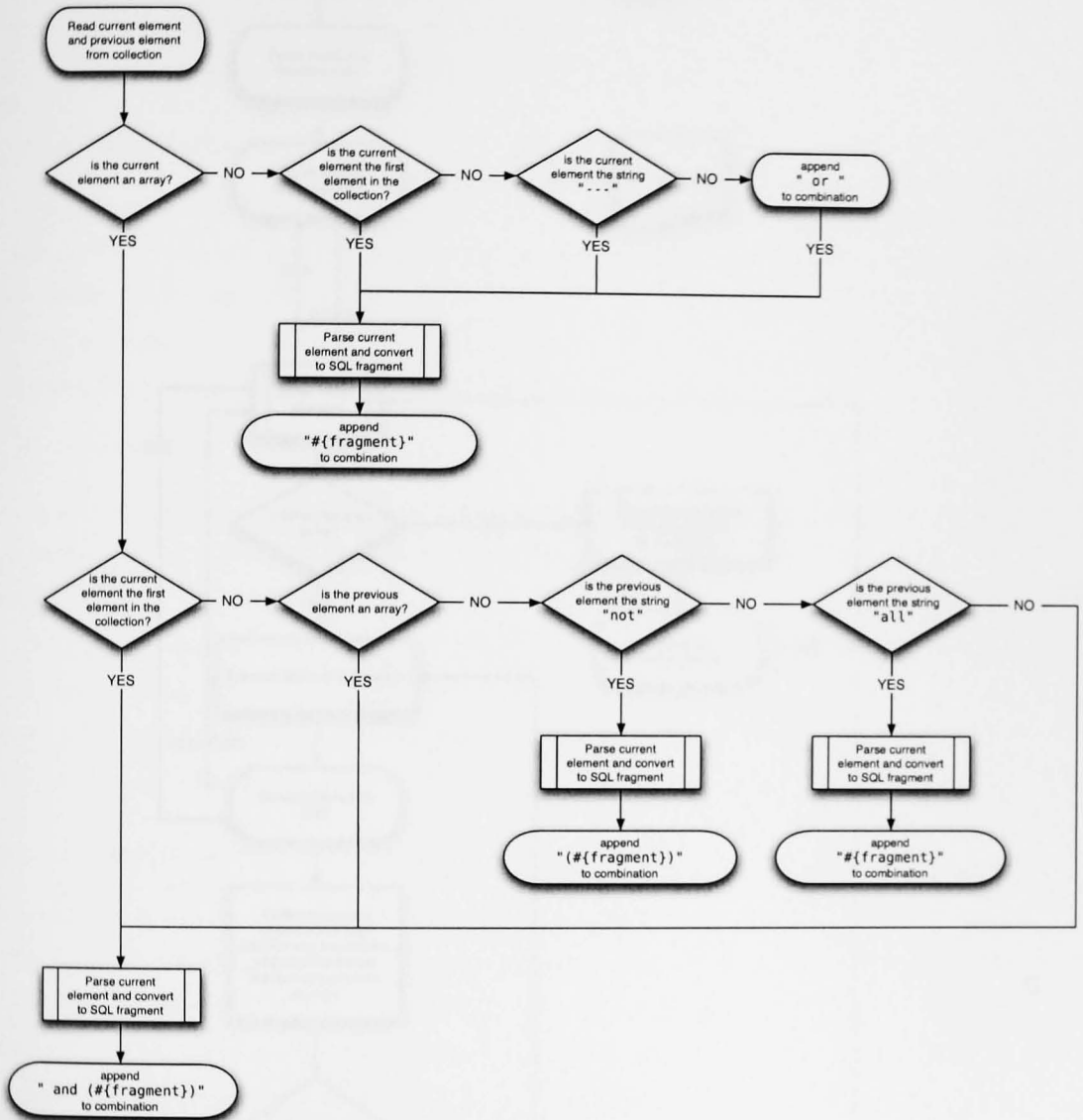


Figure 6.7: Combining YAML Elements

#### 6.3.4.2 Ruby erb Evaluation

As described in Section 2.4.1, erb is a simple templating system, which allows Ruby code to be embedded within plain text. As described in Section 6.3.3.3, and illustrated in Figure 6.5, as part of the Wish compilation process, Ruby expressions embedded within erb tags are interpreted, executed and replaced

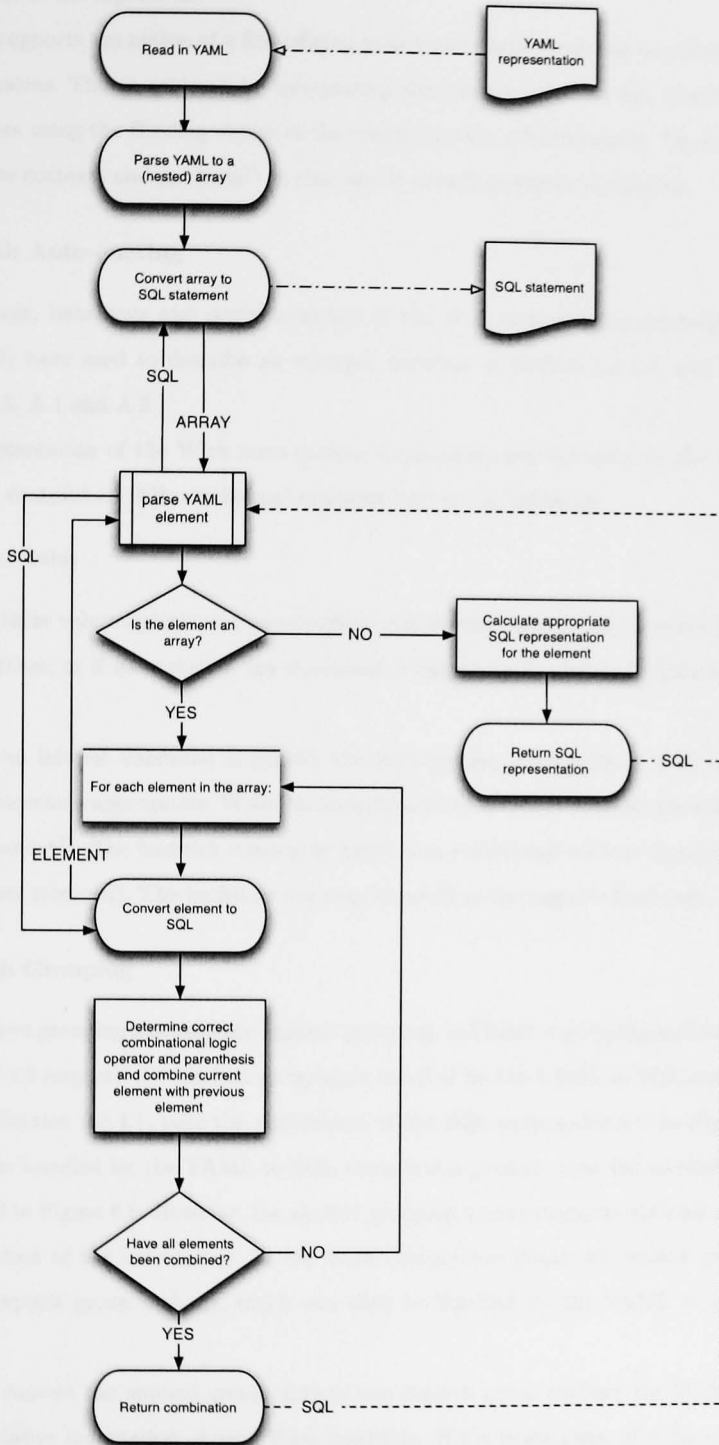


Figure 6.8: Converting YAML to SQL

with the output of the expression.

Wish also supports the notion of a file defining relative artefacts which can be referred to from within the erb expressions. This is achieved by interpreting the relative artefacts file, storing it as a Binding object, and then using the Binding object as the context for the erb evaluation. The ability to represent and manipulate contexts and methods<sup>11</sup> in this way is a truly powerful mechanism.

#### 6.3.4.3 Wish Auto-quoting

The development, behaviour and implementation of the Wish auto-quoting mechanism (described in Section 6.2.1.1) were used to describe an example iteration in Section 6.3.1.4, and is the subject of Appendices A.3, A.1 and A.2.

The implementation of the Wish auto-quoting mechanism only operates on the values of interest condition that contains an SQL conditional operator such as the following:

```
attribute = value
```

It matches these values against a series of regular expressions. Depending on which particular regular expression matches, or if none match, the statement is tagged appropriately. Figure 6.9 illustrates this process.

Each time an interest condition is parsed, the auto-quoting mechanism is used to auto-quote the condition's value where appropriate. With the exception of the backtick removal, the auto-quoting mechanism is idempotent<sup>12</sup>. The backtick removal is based on a conditional with an idempotent default (the backticks are not removed). The backticks are only removed in the parser's final pass.

#### 6.3.4.4 Wish Grouping

Wish employs two grouping mechanisms: implicit grouping, and explicit grouping as described in Sections 6.2.4.1 and 6.2.4.2 respectively. Implicit grouping is handled by the YAML to SQL compilation process as detailed in Section 6.3.4.1; note the parenthesis in the SQL output detailed in Figure 6.7. Explicit grouping is also handled by the YAML to SQL compilation process; note the parenthesis in the SQL output detailed in Figure 6.6. However, the explicit grouping syntax supports optional grouping endings through inspection of the indentation. In the Wish compilation phase, all implicit group endings are converted to explicit group endings, which can then be handled by the YAML to SQL compilation process.

In order to convert the implicit group endings into explicit group endings, the Wish parser needs to consider the relative indentation of each Wish condition. If the indentation of a line subsequent to an

<sup>11</sup>as opposed to the ability to manipulate just pointers to objects and primitives in languages such as Java (ignoring the syntactic monstrosity that is a Java anonymous inner class).

<sup>12</sup>The concept of idempotence originates from mathematics. It refers to an operation that yields the same result whether applied once, or more than once. For example, multiplying any given integer by 0 is idempotent:  $8 \times 0 \times 0 \times 0 \dots = 8 \times 0$ .



explicit group starting tag is at the same or smaller indentation, and there is no explicit group ending, an explicit group ending is inserted into the Wish statement. This is implemented by maintaining a list of the indentations of all unclosed explicit groups encountered so far. Each time the parser encounters a line with a smaller indentation, the unclosed groups list is checked, and if any groups need to be closed they are, and are subsequently removed from the unclosed groups list.

#### 6.3.4.5 Wish Abstraction

When the Wish parser encounters a condition that does not contain one of the standard SQL conditional operators listed in Table 5.2, and is not an explicit grouping marker, it assumes that it is an abstract statement referring to an external file. Wish looks for a file matching the name of the abstract statement with the extension `.wish`. The contents of this file are wrapped with explicit grouping markers (all by default, or not if the abstract statement started with the `not` keyword).

Wish implements this behaviour by reading through the Wish file, and building an internal array of strings representing each line. When the parser reaches an abstract statement, it locates the appropriate file, reads in all the lines, surrounds them with the appropriate explicit grouping markers, and inserts them in place of the abstract statement ensuring that the initial space is preserved. Once the parser has reached the end of the original file, it checks to see if it found and replaced any abstract statements. If so, it starts the whole process again in order to replace any new abstract statements. If it did not replace any abstract statements then the process is concluded.

If the abstract condition contains additional tokens separated by white-space these are assumed to be parameters to the abstraction. For each parameter, the abstract file is searched for matching parameter tokens, and all occurrences of these tokens are replaced by the parameter. This takes place before the lines are inserted into the Wish statement. The matching parameters are simply uppercase letters surrounded with pipe characters (such as `|A|`). The first parameter replaces all occurrences of `|A|`, the second parameter replaces all occurrences of `|B|`, etc.

#### 6.3.4.6 Wish Scoping

The Wish scoping mechanism uses the abstraction mechanism described in Section 6.1.2.5. However it generates a different condition to insert into the Wish statement. When the Wish parser finds an interest condition that uses the SQL `in` operator it expects that the value of that condition to be the name of an abstraction. The file is located (as described in Section 6.3.4.5) and parsed as a separate Wish file. The resulting SQL where clause is then used to construct a nested SQL statement similar to the following:

```
attribute in (select attribute from artefacts where #{convert_wish_to_sql})
```

#### **6.3.4.7 Overview**

Figure 6.10 gives an overview of the various stages of the Wish compiler.

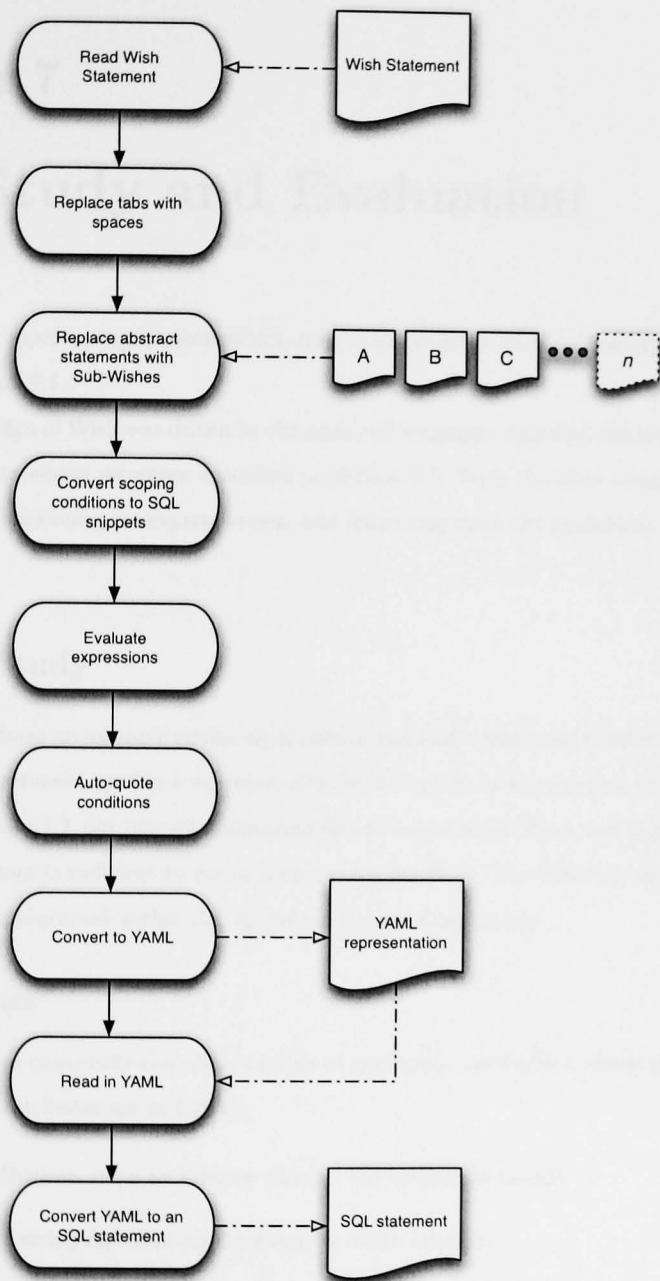


Figure 6.10: The Wish Compiler



## Chapter 7

# Case Study and Evaluation

Wish is a DSL for representing interests which aims to be modular, succinct, readable and expressive as described in Section 6.1.1

The overall design of Wish was driven by the aims and language objectives described in Section 6.1.1, and the interest statement structure described in Section 6.1. Wish therefore needed to support these structures whilst maintaining its expressiveness, and improving upon the readability and succinctness of SQL.

### 7.1 Case Study

This section introduces an example virtual environment and this is then used to illustrate the capabilities of Wish. The environment used is a snapshot of a football pitch, with players a ball and a referee. As discussed in Section 4.1.2, our interest statements do not reason about time, and therefore a snapshot of a virtual environment is sufficient to use as a case study for Wish. The following sections introduce the numerous artefacts contained within the virtual environment case study.

#### 7.1.1 Artefacts

The artefacts in this case study contain a number of additional attributes to those described in Section 4.4.1. These extra attributes are as follows:

**virtual** This is a Boolean value to indicate whether the artefact is virtual.

**category** This is a string representing the category of the artefact.

**name** This is the name of the concept that the artefact is associated with (i.e. the name of a person).

Table 7.1 lists the artefact attributes and their associated types.

Table 7.1: Artefact Attributes

Name	Type
shape	string
width	float
height	float
length	float
x_coord	float
y_coord	float
z_coord	float
radius	float
colour	string
transparency	float
id	integer
virtual	boolean
category	string
name	string

7.1.2 Football Pitch

The football pitch is comprised of a number of artefacts. These artefacts are detailed in Table 7.2. Figures 7.1 and 7.2 present two different views of the pitch.

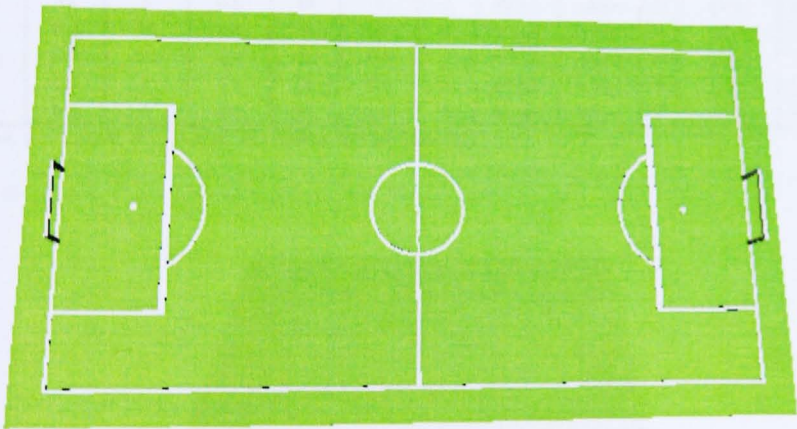


Figure 7.1: Aerial View of the Football Pitch

Figure 7.3 illustrates the location and shape of the following main areas or locales within the football pitch<sup>1</sup>:

- centre circle,

<sup>1</sup>Notice that due to the implementation of the virtual environment, the x and y axis are different from those presented in Figure 5.1.

Table 7.2: Football Pitch Artefacts

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
5	box	50	0.05	50	25	25	0	NULL	green	0	pitch	0	home half
6	box	50	0.05	50	25	75	0	NULL	green	0	pitch	0	away half
7	box	50	0.05	0.5	25	50	0.1	NULL	white	0	pitch	0	centre line
8	cylinder	NULL	0.05	NULL	25	50	0.05	7	white	0	pitch	0	centre circle outer
9	cylinder	NULL	0.05	NULL	25	50	0.1	6.5	green	0	pitch	0	centre circle
10	box	30	0.05	15	25	7.5	0.15	NULL	white	0	pitch	0	home goal outer
11	box	29	0.05	14	25	7	0.2	NULL	green	0	pitch	0	home goal
12	box	30	0.05	15	25	92.5	0.15	NULL	white	0	pitch	0	away goal outer
13	box	29	0.05	14	25	93	0.2	NULL	green	0	pitch	0	away goal
14	cylinder	NULL	0.05	NULL	25	10	0.05	10	white	0	pitch	0	home penalty circle outer
15	cylinder	NULL	0.05	NULL	25	10	0.1	9.5	green	0	pitch	0	home penalty circle
16	cylinder	NULL	0.05	NULL	25	10	0.25	0.5	white	0	pitch	0	home penalty spot
17	cylinder	NULL	0.05	NULL	25	90	0.05	10	white	0	pitch	0	away penalty circle outer
18	cylinder	NULL	0.05	NULL	25	90	0.1	9.5	green	0	pitch	0	away penalty circle
19	cylinder	NULL	0.05	NULL	25	90	0.25	0.5	white	0	pitch	0	away penalty spot
20	box	0.5	0.05	100	-0.25	50	0.1	NULL	white	0	pitch	0	near touch line
21	box	0.5	0.05	100	50.25	50	0.1	NULL	white	0	pitch	0	far touch line
22	box	51	0.05	0.5	25	-0.25	0.1	NULL	white	0	pitch	0	home touch line
23	box	51	0.05	0.5	25	100.25	0.1	NULL	white	0	pitch	0	away touch line
24	box	5	0.05	100	-2.5	50	0	NULL	green	0	pitch	0	near touch area
25	box	5	0.05	100	52.5	50	0	NULL	green	0	pitch	0	far touch area
26	box	60	0.05	5	25	-2.5	0	NULL	green	0	pitch	0	home touch area
27	box	60	0.05	5	25	102.5	0	NULL	green	0	pitch	0	away touch area
28	box	0.5	5	0.5	20	0	0	NULL	white	0	pitch	0	home near goal post
29	box	0.5	5	0.5	30	0	0	NULL	white	0	pitch	0	home far goal post
30	box	10.5	0.5	0.5	25	0	5	NULL	white	0	pitch	0	home goal crossbar
31	box	10.5	0.5	0.5	25	100	5	NULL	white	0	pitch	0	away goal crossbar
32	box	0.5	5	0.5	20	100	0	NULL	white	0	pitch	0	away near goal post
33	box	0.5	5	0.5	30	100	0	NULL	white	0	pitch	0	away far goal post

(x = x coord, y = y coord, z = z coord, v = virtual, t = transparency)



Figure 7.2: Stadium View of the Football Pitch

- home and away halves,
- home and away goal areas,
- home and away penalty circles.

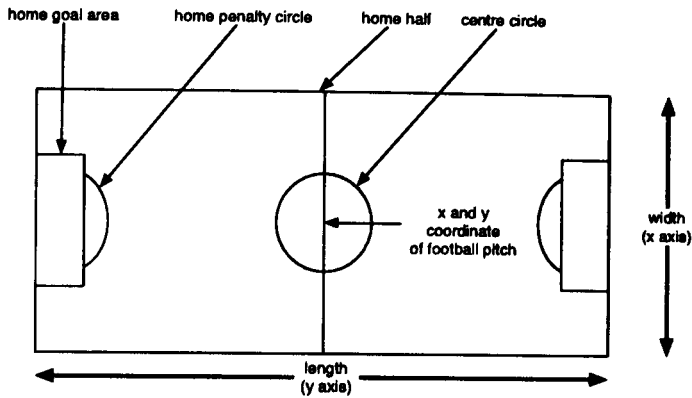


Figure 7.3: Football Pitch Areas

### 7.1.3 Players

In addition to the pitch, as described in Section 7.1.2, the case study also includes a number of players split into two teams: the red team and the blue team. Table 7.3 in Appendix B describes the artefacts that represent the red team and they are illustrated in Figure 7.4.

Table 7.3: All Red Players

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
34	cone	NULL	3	NULL	30	48	0	1	red	0	player	0	sam
36	cone	NULL	3	NULL	50	38	0	1	red	0	player	0	bob
38	cone	NULL	3	NULL	10	8	0	1	red	0	player	0	john
40	cone	NULL	3	NULL	30	12	0	1	red	0	player	0	jim
42	cone	NULL	3	NULL	40	70	0	1	red	0	player	0	geoffrey
44	cone	NULL	3	NULL	40	90	0	1	red	0	player	0	bernard
46	cone	NULL	3	NULL	20	30	0	1	red	0	player	0	toddy
48	cone	NULL	3	NULL	40	43	0	1	red	0	player	0	charlie
50	cone	NULL	3	NULL	20	60	0	1	red	0	player	0	rupert
52	cone	NULL	3	NULL	5	20	0	1	red	0	player	0	even

(x = x coord, y = y coord, z = z coord, v = virtual, t = transparency)

Table 7.4 describes the artefacts that represent the blue team. They are illustrated in Figure 7.5.



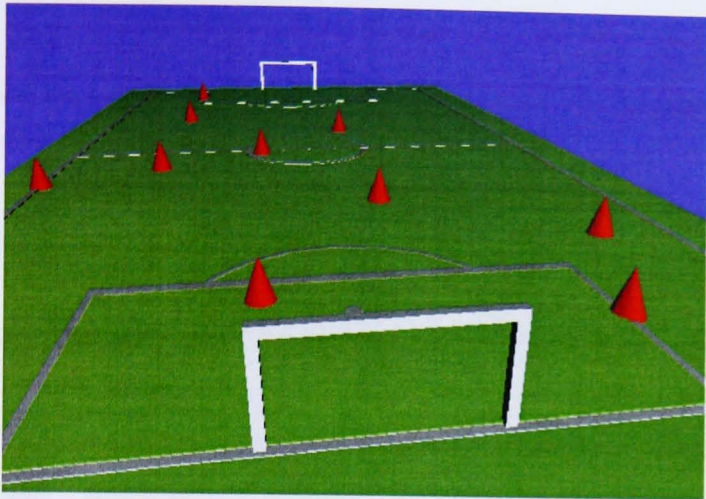


Figure 7.4: The Red Team

Table 7.4: All Blue Players

Id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
56	cone	NULL	3	NULL	30	60	0	1	blue	0	player	0	karl
58	cone	NULL	3	NULL	35	45	0	1	blue	0	player	0	hendrik
60	cone	NULL	3	NULL	15	13	0	1	blue	0	player	0	david
62	cone	NULL	3	NULL	45	85	0	1	blue	0	player	0	han
64	cone	NULL	3	NULL	35	14	0	1	blue	0	player	0	jean
66	cone	NULL	3	NULL	45	55	0	1	blue	0	player	0	boris
68	cone	NULL	3	NULL	35	95	0	1	blue	0	player	0	bilbo
70	cone	NULL	3	NULL	45	65	0	1	blue	0	player	0	savas
72	cone	NULL	3	NULL	25	15	0	1	blue	0	player	0	chris

(*x* = *x* coord, *y* = *y* coord, *z* = *z* coord, *v* = *virtual*, *t* = *transparency*)

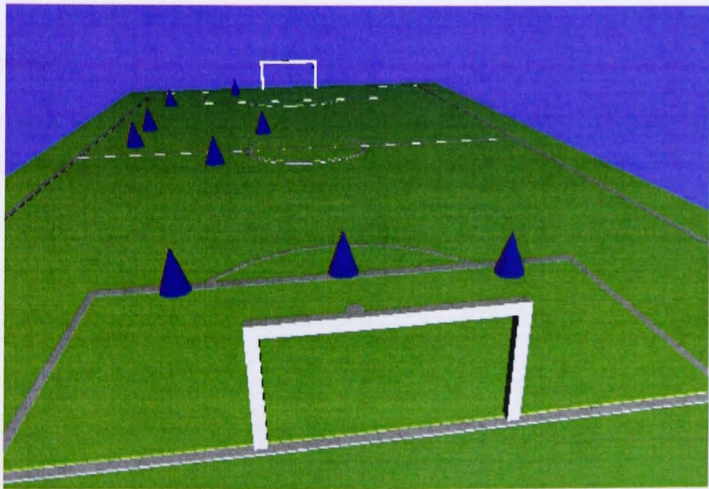


Figure 7.5: The Blue Team

Each team also has a goalkeeper. Table 7.5 describes the artefacts that represent the goalkeepers. These are illustrated in Figure 7.6.

Table 7.5: Goalkeepers

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
54	cone	NULL	3	NULL	26	0	0	1	yellow	0	goalie	0	tim
74	cone	NULL	3	NULL	25	100	0	1	yellow	0	goalie	0	carlos

(*x* = *x* coord, *y* = *y* coord, *z* = *z* coord, *v* = *virtual*, *t* = *transparency*)

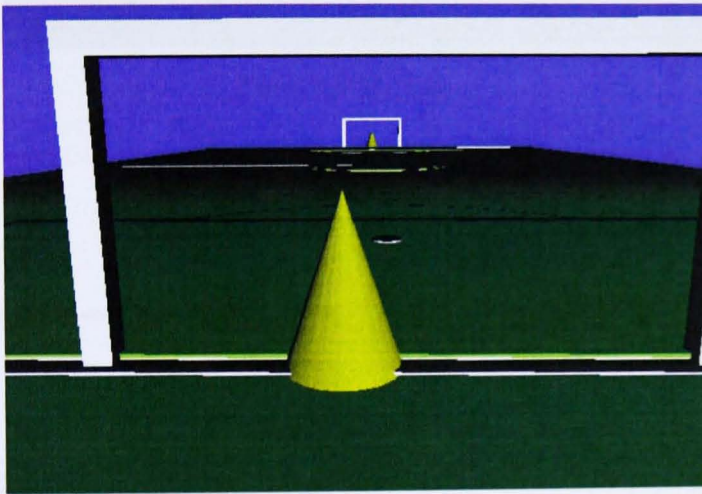


Figure 7.6: The Goalkeepers

#### 7.1.4 Referee

Controlling the game is a referee<sup>2</sup>. Table 7.6 describes the artefact that represents the referee which is then illustrated in Figure 7.7.

Table 7.6: The Referee

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
76	cone	NULL	3	NULL	27	13	0	1	turquoise	0	player	0	ref

(*x* = *x* coord, *y* = *y* coord, *z* = *z* coord, *v* = *virtual*, *t* = *transparency*)

<sup>2</sup>Susanna spake, and the referee became turquoise. It was chosen.

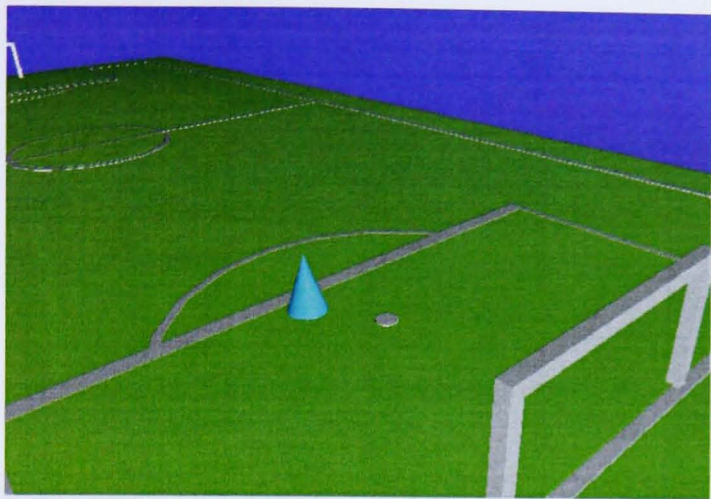


Figure 7.7: The Referee

7.1.5 Football

The football pitch also contains a football. Table 7.7 describes this artefact which is also illustrated in Figure 7.8.

Table 7.7: The Football

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
78	sphere	NULL	NULL	NULL	15	7	2	0.5	white	0	ball	0	ball

(*x* = *x* coord, *y* = *y* coord, *z* = *z* coord, *v* = *virtual*, *t* = *transparency*)

7.1.6 Locales

The virtual environment also contains a virtual artefact representing a locale. The area of the locale is the near side of the football pitch. Table 7.8 describes this artefact and is illustrated in Figure 7.9.

Table 7.8: Locales

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
79	box	25	0.05	100	12.5	50	1	NULL	brown	1	locale	0.5	near half

(*x* = *x* coord, *y* = *y* coord, *z* = *z* coord, *v* = *virtual*, *t* = *transparency*)



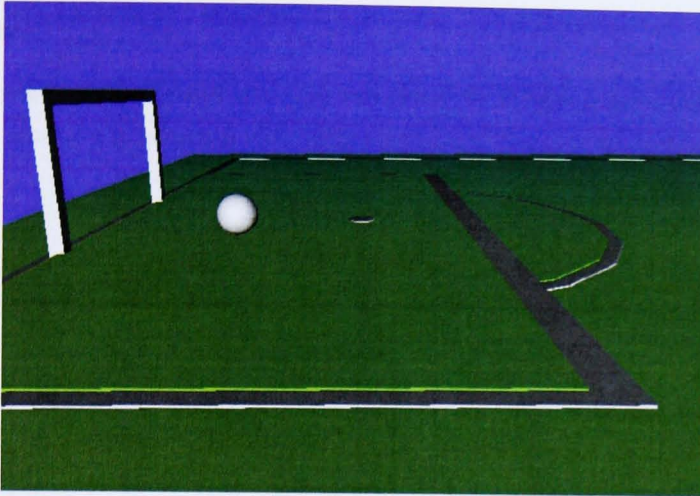


Figure 7.8: The Football

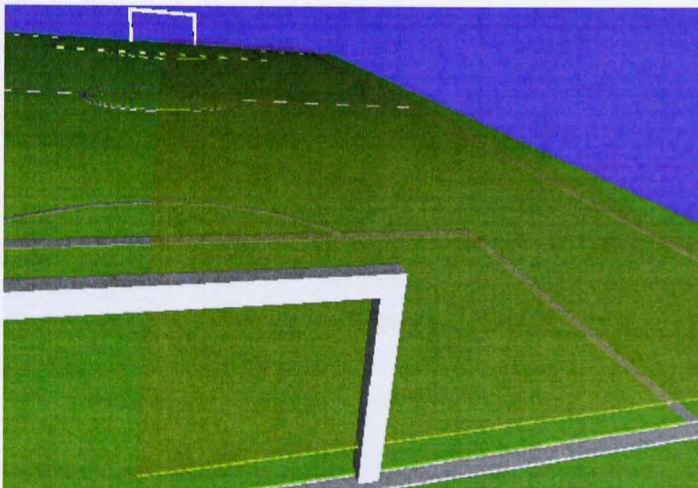


Figure 7.9: The Locale Representing the Near Half of the Pitch



### 7.1.7 Auras

Each of the players has an associated aura. The aura shares the same x, y and z coordinates and also the name of the artefact it is associated with. Table 7.9 describes these artefacts and they are illustrated in Figure 7.10.

Table 7.9: Auras

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
35	cylinder	NULL	0.5	NULL	30	48	0	5	red	1	aura	0.5	sam
37	cylinder	NULL	0.5	NULL	50	38	0	5	red	1	aura	0.5	bob
39	cylinder	NULL	0.5	NULL	10	8	0	5	red	1	aura	0.5	john
41	cylinder	NULL	0.5	NULL	30	12	0	5	red	1	aura	0.5	jim
43	cylinder	NULL	0.5	NULL	40	70	0	5	red	1	aura	0.5	geoffrey
45	cylinder	NULL	0.5	NULL	40	90	0	5	red	1	aura	0.5	bernard
47	cylinder	NULL	0.5	NULL	20	30	0	5	red	1	aura	0.5	toddy
49	cylinder	NULL	0.5	NULL	40	43	0	5	red	1	aura	0.5	charlie
51	cylinder	NULL	0.5	NULL	20	60	0	5	red	1	aura	0.5	rupert
53	cylinder	NULL	0.5	NULL	5	20	0	5	red	1	aura	0.5	even
55	cylinder	NULL	0.5	NULL	26	0	0	5	red	1	aura	0.5	tim
57	cylinder	NULL	0.5	NULL	30	60	0	5	red	1	aura	0.5	karl
59	cylinder	NULL	0.5	NULL	35	45	0	5	red	1	aura	0.5	hendrik
61	cylinder	NULL	0.5	NULL	15	13	0	5	red	1	aura	0.5	david
63	cylinder	NULL	0.5	NULL	45	85	0	5	red	1	aura	0.5	han
65	cylinder	NULL	0.5	NULL	35	14	0	5	red	1	aura	0.5	jean
67	cylinder	NULL	0.5	NULL	45	55	0	5	red	1	aura	0.5	boris
69	cylinder	NULL	0.5	NULL	35	95	0	5	red	1	aura	0.5	bilbo
71	cylinder	NULL	0.5	NULL	45	65	0	5	red	1	aura	0.5	savas
73	cylinder	NULL	0.5	NULL	25	15	0	5	red	1	aura	0.5	chris
75	cylinder	NULL	0.5	NULL	25	100	0	5	red	1	aura	0.5	carlos
77	cylinder	NULL	0.5	NULL	27	13	0	5	red	1	aura	0.5	ref

(x = x coord, y = y coord, z = z coord, v = virtual, t = transparency)

### 7.1.8 All Artefacts

Tables 7.10 and 7.11 present all of the virtual environment artefacts. These artefacts are also illustrated in Figures 7.11 and 7.12.

## 7.2 Example Statements

This section will revisit the examples introduced in Section 3.3 within the context of the Wish syntax as defined in Section 6.2. In the following examples, any subwishes or relative artefact declarations are presented inline. However, a complete version of the `relative_artefacts.rb` file is presented in Section C.1,

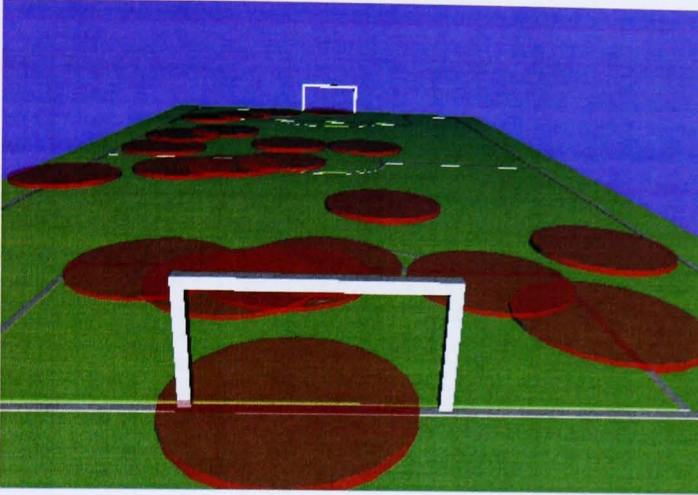


Figure 7.10: All the Auras



Figure 7.11: A Stadium View of All Artefacts

Table 7.10: All Artefacts

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
5	box	50	0.05	50	25	25	0	NULL	green	0	pitch	0	home half
6	box	50	0.05	50	25	75	0	NULL	green	0	pitch	0	away half
7	box	50	0.05	0.5	25	50	0.1	NULL	white	0	pitch	0	centre line
8	cylinder	NULL	0.05	NULL	25	50	0.05	7	white	0	pitch	0	centre circle outer
9	cylinder	NULL	0.05	NULL	25	50	0.1	6.5	green	0	pitch	0	centre circle
10	box	30	0.05	15	25	7.5	0.15	NULL	white	0	pitch	0	home goal outer
11	box	29	0.05	14	25	7	0.2	NULL	green	0	pitch	0	home goal
12	box	30	0.05	15	25	92.5	0.15	NULL	white	0	pitch	0	away goal outer
13	box	29	0.05	14	25	93	0.2	NULL	green	0	pitch	0	away goal
14	cylinder	NULL	0.05	NULL	25	10	0.05	10	white	0	pitch	0	home penalty circle outer
15	cylinder	NULL	0.05	NULL	25	10	0.1	9.5	green	0	pitch	0	home penalty circle
16	cylinder	NULL	0.05	NULL	25	10	0.25	0.5	white	0	pitch	0	home penalty spot
17	cylinder	NULL	0.05	NULL	25	90	0.05	10	white	0	pitch	0	away penalty circle outer
18	cylinder	NULL	0.05	NULL	25	90	0.1	9.5	green	0	pitch	0	away penalty circle
19	cylinder	NULL	0.05	NULL	25	90	0.25	0.5	white	0	pitch	0	away penalty spot
20	box	0.5	0.05	100	-0.25	50	0.1	NULL	white	0	pitch	0	near touch line
21	box	0.5	0.05	100	50.25	50	0.1	NULL	white	0	pitch	0	far touch line
22	box	51	0.05	0.5	25	-0.25	0.1	NULL	white	0	pitch	0	home touch line
23	box	51	0.05	0.5	25	100.25	0.1	NULL	white	0	pitch	0	away touch line
24	box	5	0.05	100	-2.5	50	0	NULL	green	0	pitch	0	near touch area
25	box	5	0.05	100	52.5	50	0	NULL	green	0	pitch	0	far touch area
26	box	60	0.05	5	25	-2.5	0	NULL	green	0	pitch	0	home touch area
27	box	60	0.05	5	25	102.5	0	NULL	green	0	pitch	0	away touch area
28	box	0.5	5	0.5	20	0	0	NULL	white	0	pitch	0	home near goal post
29	box	0.5	5	0.5	30	0	0	NULL	white	0	pitch	0	home far goal post
30	box	10.5	0.5	0.5	25	0	5	NULL	white	0	pitch	0	home goal crossbar
31	box	10.5	0.5	0.5	25	100	5	NULL	white	0	pitch	0	away goal crossbar
32	box	0.5	5	0.5	20	100	0	NULL	white	0	pitch	0	away near goal post
33	box	0.5	5	0.5	30	100	0	NULL	white	0	pitch	0	away far goal post
34	cone	NULL	3	NULL	30	48	0	1	red	0	player	0	sam
35	cylinder	NULL	0.5	NULL	30	48	0	5	red	1	aura	0.5	sam
36	cone	NULL	3	NULL	50	38	0	1	red	0	player	0	bob
37	cylinder	NULL	0.5	NULL	50	38	0	5	red	1	aura	0.5	bob
38	cone	NULL	3	NULL	10	8	0	1	red	0	player	0	john
39	cylinder	NULL	0.5	NULL	10	8	0	5	red	1	aura	0.5	john
40	cone	NULL	3	NULL	30	12	0	1	red	0	player	0	jim
41	cylinder	NULL	0.5	NULL	30	12	0	5	red	1	aura	0.5	jim
42	cone	NULL	3	NULL	40	70	0	1	red	0	player	0	geoffrey
43	cylinder	NULL	0.5	NULL	40	70	0	5	red	1	aura	0.5	geoffrey
44	cone	NULL	3	NULL	40	90	0	1	red	0	player	0	bernard
45	cylinder	NULL	0.5	NULL	40	90	0	5	red	1	aura	0.5	bernard
46	cone	NULL	3	NULL	20	30	0	1	red	0	player	0	toddy
47	cylinder	NULL	0.5	NULL	20	30	0	5	red	1	aura	0.5	toddy
48	cone	NULL	3	NULL	40	43	0	1	red	0	player	0	charlie
49	cylinder	NULL	0.5	NULL	40	43	0	5	red	1	aura	0.5	charlie
50	cone	NULL	3	NULL	20	60	0	1	red	0	player	0	rupert

Continued in Table B.10

Table 7.11: All Artefacts (Continued from Table B.9)

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
51	cylinder	NULL	0.5	NULL	20	60	0	5	red	1	aura	0.5	rupert
52	cone	NULL	3	NULL	5	20	0	1	red	0	player	0	sven
53	cylinder	NULL	0.5	NULL	5	20	0	5	red	1	aura	0.5	sven
54	cone	NULL	3	NULL	26	0	0	1	yellow	0	goalie	0	tim
55	cylinder	NULL	0.5	NULL	26	0	0	5	red	1	aura	0.5	tim
56	cone	NULL	3	NULL	30	60	0	1	blue	0	player	0	karl
57	cylinder	NULL	0.5	NULL	30	60	0	5	red	1	aura	0.5	karl
58	cone	NULL	3	NULL	35	45	0	1	blue	0	player	0	hendrik
59	cylinder	NULL	0.5	NULL	35	45	0	5	red	1	aura	0.5	hendrik
60	cone	NULL	3	NULL	15	13	0	1	blue	0	player	0	david
61	cylinder	NULL	0.5	NULL	15	13	0	5	red	1	aura	0.5	david
62	cone	NULL	3	NULL	45	85	0	1	blue	0	player	0	han
63	cylinder	NULL	0.5	NULL	45	85	0	5	red	1	aura	0.5	han
64	cone	NULL	3	NULL	35	14	0	1	blue	0	player	0	jean
65	cylinder	NULL	0.5	NULL	35	14	0	5	red	1	aura	0.5	jean
66	cone	NULL	3	NULL	45	55	0	1	blue	0	player	0	boris
67	cylinder	NULL	0.5	NULL	45	55	0	5	red	1	aura	0.5	boris
68	cone	NULL	3	NULL	35	95	0	1	blue	0	player	0	bilbo
69	cylinder	NULL	0.5	NULL	35	95	0	5	red	1	aura	0.5	bilbo
70	cone	NULL	3	NULL	45	65	0	1	blue	0	player	0	savas
71	cylinder	NULL	0.5	NULL	45	65	0	5	red	1	aura	0.5	savas
72	cone	NULL	3	NULL	25	15	0	1	blue	0	player	0	chris
73	cylinder	NULL	0.5	NULL	25	15	0	5	red	1	aura	0.5	chris
74	cone	NULL	3	NULL	25	100	0	1	yellow	0	goalie	0	carlos
75	cylinder	NULL	0.5	NULL	25	100	0	5	red	1	aura	0.5	carlos
76	cone	NULL	3	NULL	27	13	0	1	turquoise	0	player	0	ref
77	cylinder	NULL	0.5	NULL	27	13	0	5	red	1	aura	0.5	ref
78	sphere	NULL	NULL	NULL	15	7	2	0.5	white	0	ball	0	ball
79	box	25	0.05	100	12.5	50	1	NULL	brown	1	locale	0.5	near half

(x = x coord, y = y coord, z = z coord, v = virtual, t = transparency)

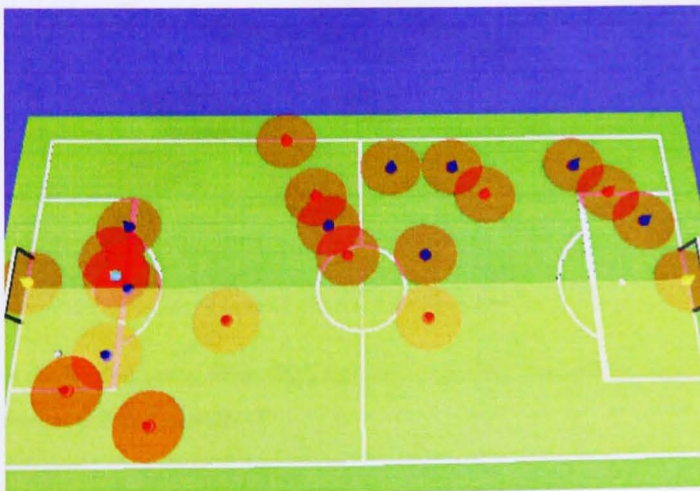


Figure 7.12: An Aerial View of All Artefacts

and all of the subwish listings are presented in Appendix D.

### 7.2.1 Categories

As an example of categories, Section 3.3.4 described the interesting set consisting of all red artefacts. This type of statement is represented in terms of the relationship between the statement's given values and the artefact's current attribute values. An English version of this statement could be:

*I am interested in all artefacts that are red*

Wish automatically assumes that the statement is about interest, and so focusses purely on the condition:

```
coloured red
```

The above Wish statement used a subwish called `coloured.wish` which provides a more readable version than the SQL-like equivalent:

```
#coloured.wish
colour = |A|
```

Figure 7.13 provides a view of the virtual environment given the above interests. Notice that as the pitch is not red it does not appear in the set of interesting artefacts. The red artefacts therefore appear to be floating in space. The following example statements will assume that the user is additionally interested in the football pitch for the purpose of providing a context and setting for the interesting artefacts.

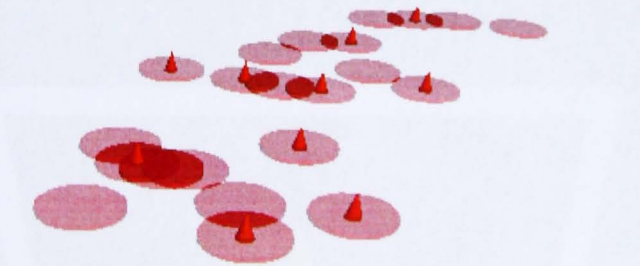


Figure 7.13: All Red Artefacts

A summary of the English prose, Wish, SQL and set of matching artefacts for this example statement is presented in Section C.2 in Appendix C.

### 7.2.2 Locales

An example of a statement using locales is as follows:



*I am interested in all artefacts within the near half of the pitch*

Given the existence of a virtual artefact representing the area of the near half of the pitch, we can declare a variable within the `relative_artefacts.rb` file:

```
near_half = find_locale 'near half'
```

This variable can then be used within a wish statement as follows<sup>3</sup>:

```
category = pitch
within_box near_half
```

The subwish `within_box` uses the algorithm described in Section 5.3.2, which in Wish looks as follows:

```
#within_box.wish
x_coord >= <%= |A|.x_coord - (|A|.width / 2)%>
x_coord <= <%= |A|.x_coord + (|A|.width / 2)%>
y_coord >= <%= |A|.y_coord - (|A|.length / 2)%>
y_coord <= <%= |A|.y_coord + (|A|.length / 2)%>
id != <%= |A|.id%>
```

Figure 7.14 provides a view of all the artefacts within the near half.

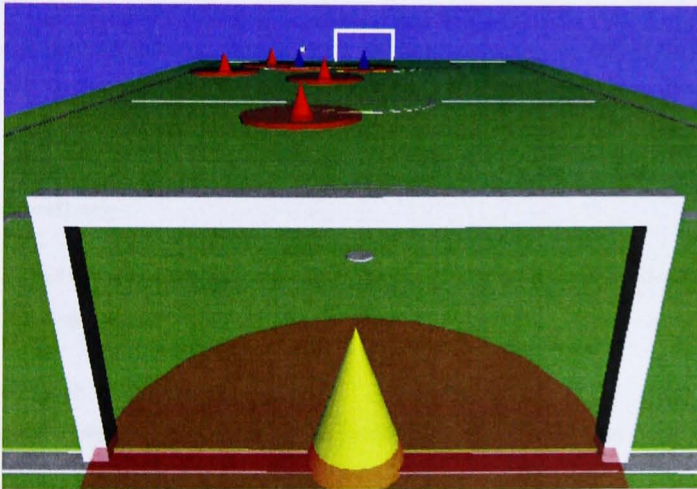


Figure 7.14: All Artefacts Within the Near Half of the Pitch

A summary of the English prose, Wish, SQL and set of matching artefacts for this example statement is presented in Section C.3 in Appendix C.

<sup>3</sup>Notice that the condition `category = pitch` is added purely to provide visual context for the images such as that presented in Figure 7.14.

### 7.2.3 Relative Locales

As described in Section 5.2.3, in terms of interest statements, relative locales can be treated identically to standard locales as they are both represented by relative artefacts. The section also discussed that any relationship between artefacts should be resolved in the declaration of the relative artefacts. As an example of this consider the following interest statement:

*I am interested in all artefacts within the referee's aura*

Here the interest is relative to the referee's aura. We have a relationship between an aura and the referee. This relationship should be resolved in the declaration of the relative artefacts. Consider the following snippet from `relative_artefacts.rb`:

```
ref_aura = find_aura 'ref'
```

The declaration of this relative artefact uses the method `find_aura` which is defined as follows:

```
def find_aura(name)
  Artefact.find(:first, :conditions => {:name => name, :virtual => true, :category => 'aura'})
end
```

This method relies on the fact that the aura shares the same name as the concept it is representing. For example, Bob's aura is named Bob. However, to distinguish the two, the aura has the category of aura. The `find_aura` method filters on this category, implicitly creating the relationship.

Given the newly declared relative artefact, we can use it in a Wish statement as follows:

```
category = pitch
within_circle ref_aura
```

The `within_circle` subwish is implemented with the algorithm introduced in Section 5.3.4 which can be defined with Wish as follows:

```
#within_circle.wish
<=>|A|.radius%> > `sqrt(pow((x_coord - <=> |A|.x_coord%>), 2) + pow((y_coord - <=> |A|.y_coord%>), 2))`
```

Figure 7.15 provides a view of all the artefacts within the referee's aura.

A summary of the English prose, Wish, SQL and set of matching artefacts for this example statement is presented in Section C.4 in Appendix C.

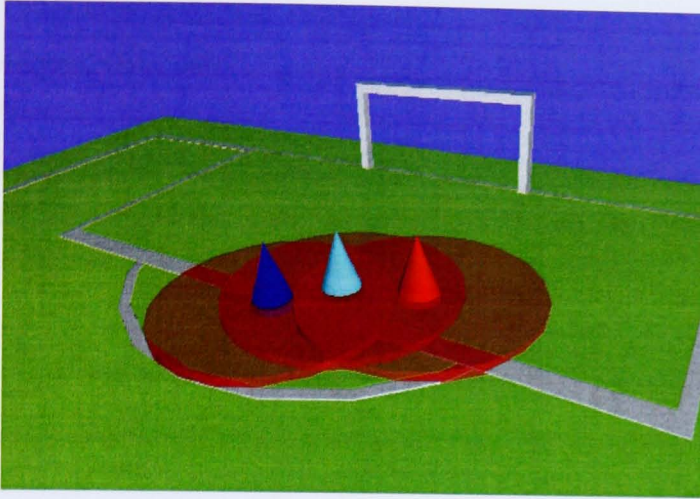


Figure 7.15: Artefacts Within the Referee's Aura

## 7.2.4 Interacting Locales

Section 3.3.3 described an example of interacting locales as follows

*“When an artefact A’s aura collides with artefact B’s aura, artefact A can be said to be aware of artefact B.”*

In terms of the interest concepts, as defined in Section 5.1, concepts such as the one above can be represented by introducing two relative artefacts into the interest statement. For example:

*I am interested in all artefacts whose aura overlaps the referee’s aura*

Section 7.2.3 described how to define the relative artefact `ref_aura`. With such a variable declared it’s possible to define the following Wish statement:

```
category = pitch
in_awareness_range_of ref_aura
```

The definition of the subwish `in_awareness_range_of.wish` is as follows:

```
#in_awareness_range_of.wish
virtual = false
name in auras_in_awareness_range_of |A|
```

Here we are describing an interest in non-virtual artefacts that have the same name as the set of auras in awareness range of the referee’s aura. This is achieved by scoping the name with the subwish `auras_in_awareness_range_of` which is defined as follows:



```
#auras_in_awareness_range_of.wish

overlaps |A|

category = aura
```

This statement describes auras that overlap the referee's aura. The definition of `overlaps.wish` is based on the algorithm described in Section 5.3.3 and is represented in Wish as follows:

```
#overlaps.wish

<%=|A|.radius%> + radius > `sqrt(pow((x_coord - <%= |A|.x_coord%>), 2) + pow((y_coord - <%= |A|.y_coord%>), 2))`
```

Figure 7.16 gives a view of all the artefacts, and Figure 7.17 gives the same view but with the interests applied.



Figure 7.16: All Artefacts

A summary of the English prose, Wish, SQL and set of matching artefacts for this example statement is presented in Section C.5 in Appendix C.

### 7.2.5 Combinations

We might wish to be able to create Wish statements that arbitrarily combine the examples above. For example, consider the following interest statement:

*I am interested in all non-virtual artefacts that are red players, whose aura overlaps the referee's aura and that are within the home penalty circle*

As Section 6.2.3 introduced, Wish supports the logical operators `or`, `and` to combine statements, `not` to negate a statement and the grouping keywords `all`, `not`, and `---`. The above English statement is

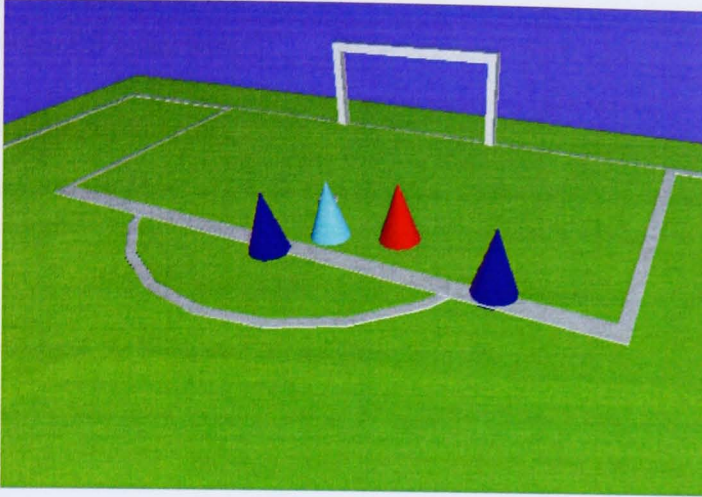


Figure 7.17: All Artefacts that are in Awareness Range of the Referee

represented in Wish as follows<sup>4</sup>:

```
category = pitch
virtual = false
coloured red
category = player
in_awareness_range_of ref_aura
within_circle home_penalty_circle
```

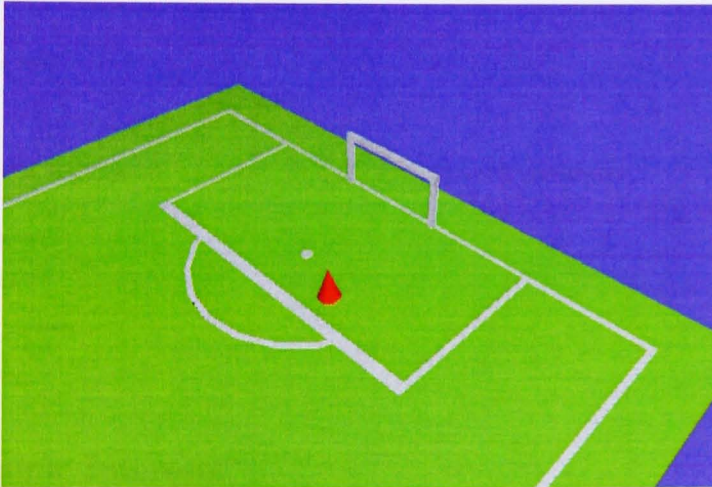


Figure 7.18: Results of a Combination Statement

<sup>4</sup>The definitions of the subwishes `coloured`, `in_awareness_range_of`, and `within_circle` are found in Sections 7.2.1, 7.2.4, and 7.2.3 respectively. Also, Appendix D contains complete listings of all subwishes used in this thesis.

A summary of the English prose, Wish, SQL and set of matching artefacts for this example statement is presented in Section C.6 in Appendix C.

## 7.2.6 Combining Concerns

Section 3.4.2 introduced the following as a method of combining the concerns of a user and the system:

$$\text{Interesting Artefacts} = ((\text{UPOS} - \text{UNEG}) \cup \text{SPOS}) - \text{SNEG} \quad (7.1)$$

Section 5.4 introduced the following SQL representation of that combination:

```
select * from Artefacts where (((id in (select id from Artefacts where UPOS) and
                                not id in (select id from Artefacts where UNEG)) or
                                id in (select id from Artefacts where SPOS)) and
                                not id in (select id from Artefacts where SNEG))
```

Wish facilitates the representation of derived sets (as described in Section 3.2.4.1) through the concept of subwishes. If we assume that the derived sets UPOS, UNEG, SPOS and SNEG have the following corresponding subwishes: `upos.wish`, `uneg.wish`, `spos.wish`, and `sneg.wish`, then we can combine these statements with `Wish` as follows:

```
all
  id in upos
  not id in uneg
  id in spos
  ...
  not id in sneg
```

Or the following semantically identical, yet more succinct `Wish` statement:

```
not id in sneg
  id in upos
    not id in uneg
  id in spos
```

## 7.3 Dynamic Interests

Section 2.6.2.2 described a number of motivations that require the ability to change interests which was one of the main factors behind the design of the dynamic interest management framework and virtual

environment axioms presented in Chapter 3 and implemented in Chapter 4. This section will illustrate how changing the interests within the virtual environment is possible if the implementation of the virtual environment follows the design presented in Chapter 4.

Consider the following Wish statement:

```
coloured turquoise
```

This statement matches the set of artefacts (just the referee) described in Table 7.12. In terms of the messages sent from the server to the client (given the architecture illustrated in Figure 4.3), the server sends the following message:

```
[{:command=>"add", :parameters=>
  {:y_coord=>13.0, :shape=>"cone", :colour=>"turquoise", :transparency=>0.0, :height=>3.0,
   :z_coord=>0.0, :id=>76, :radius=>1.0, :x_coord=>27.0}}]
```

This message indicates that the referee should be added to the client's view (as it is in the set of interesting artefacts), and sends the appropriate attributes for the view<sup>5</sup>.

Table 7.12: Turquoise Artefacts

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
76	cone	NULL	3	NULL	27	13	0	1	turquoise	0	player	0	ref

(*x = x coord, y = y coord, z = z coord, v = virtual, t = transparency*)

Now, consider that we wish to change our interest from turquoise artefacts to blue artefacts. We change our Wish statement to the following, and refresh the server:

```
coloured blue
```

The set of matching artefacts for the above statement is presented in Table 7.13. The server calculates the differences between the client's current view, and the new set of interesting artefacts and sends the following set of messages:

```
[{:command=>"delete", :parameters=>{:id=>76}},
{:command=>"add", :parameters=>
  {:y_coord=>60.0, :shape=>"cone", :colour=>"blue", :transparency=>0.0, :height=>3.0,
   :z_coord=>0.0, :id=>56, :radius=>1.0, :x_coord=>30.0}},
{:command=>"add", :parameters=>
```

<sup>5</sup>Notice that the following attributes are not sent to the client: virtual, category and name.

```

{:y_coord=>45.0, :shape=>"cone", :colour=>"blue", :transparency=>0.0, :height=>3.0,
 :z_coord=>0.0, :id=>58, :radius=>1.0, :x_coord=>35.0}},

{:command=>"add", :parameters=>
  {:y_coord=>13.0, :shape=>"cone", :colour=>"blue", :transparency=>0.0, :height=>3.0,
   :z_coord=>0.0, :id=>60, :radius=>1.0, :x_coord=>15.0}},

{:command=>"add", :parameters=>
  {:y_coord=>85.0, :shape=>"cone", :colour=>"blue", :transparency=>0.0, :height=>3.0,
   :z_coord=>0.0, :id=>62, :radius=>1.0, :x_coord=>45.0}},

{:command=>"add", :parameters=>
  {:y_coord=>14.0, :shape=>"cone", :colour=>"blue", :transparency=>0.0, :height=>3.0,
   :z_coord=>0.0, :id=>64, :radius=>1.0, :x_coord=>35.0}},

{:command=>"add", :parameters=>
  {:y_coord=>55.0, :shape=>"cone", :colour=>"blue", :transparency=>0.0, :height=>3.0,
   :z_coord=>0.0, :id=>66, :radius=>1.0, :x_coord=>45.0}},

{:command=>"add", :parameters=>
  {:y_coord=>95.0, :shape=>"cone", :colour=>"blue", :transparency=>0.0, :height=>3.0,
   :z_coord=>0.0, :id=>68, :radius=>1.0, :x_coord=>35.0}},

{:command=>"add", :parameters=>
  {:y_coord=>65.0, :shape=>"cone", :colour=>"blue", :transparency=>0.0, :height=>3.0,
   :z_coord=>0.0, :id=>70, :radius=>1.0, :x_coord=>45.0}},

{:command=>"add", :parameters=>
  {:y_coord=>15.0, :shape=>"cone", :colour=>"blue", :transparency=>0.0, :height=>3.0,
   :z_coord=>0.0, :id=>72, :radius=>1.0, :x_coord=>25.0}}]

```

Notice how the server is requesting that the referee is to be deleted (it is not blue), and the details of the blue artefacts are sent to be added to the view.

Finally consider the following statement that represents another change of interests:

```
within_circle home_penalty_circle
```

Table 7.13: Blue Artefacts

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
56	cone	NULL	3	NULL	30	60	0	1	blue	0	player	0	karl
58	cone	NULL	3	NULL	35	45	0	1	blue	0	player	0	hendrik
60	cone	NULL	3	NULL	15	13	0	1	blue	0	player	0	david
62	cone	NULL	3	NULL	45	85	0	1	blue	0	player	0	han
64	cone	NULL	3	NULL	35	14	0	1	blue	0	player	0	jean
66	cone	NULL	3	NULL	45	55	0	1	blue	0	player	0	boris
68	cone	NULL	3	NULL	35	95	0	1	blue	0	player	0	bilbo
70	cone	NULL	3	NULL	45	65	0	1	blue	0	player	0	savas
72	cone	NULL	3	NULL	25	15	0	1	blue	0	player	0	chris

(*x = x coord, y = y coord, z = z coord, v = virtual, t = transparency*)

The matching set of artefacts for the above statement is presented in Table 7.14. Again, the server calculates and sends just the differences. Most of the blue artefacts are requested to be deleted, except for one which matches the interest criteria. Details of the rest of the artefacts that match the interest criteria are sent to the client for them to be added to the view.

```
{:command=>"delete", :parameters=>{:id=>60}},
{:command=>"delete", :parameters=>{:id=>66}},
{:command=>"delete", :parameters=>{:id=>56}},
{:command=>"delete", :parameters=>{:id=>62}},
{:command=>"delete", :parameters=>{:id=>68}},
{:command=>"delete", :parameters=>{:id=>58}},
{:command=>"delete", :parameters=>{:id=>64}},
{:command=>"delete", :parameters=>{:id=>70}},

{:command=>"add", :parameters=>
{:y_coord=>7.5, :shape=>"box", :colour=>"white", :width=>30.0, :transparency=>0.0,
:height=>0.05, :length=>15.0, :z_coord=>0.15, :id=>10, :x_coord=>25.0}},

{:command=>"add", :parameters=>
{:y_coord=>7.0, :shape=>"box", :colour=>"green", :width=>29.0, :transparency=>0.0,
:height=>0.05, :length=>14.0, :z_coord=>0.2, :id=>11, :x_coord=>25.0}},

{:command=>"add", :parameters=>
{:y_coord=>10.0, :shape=>"cylinder", :colour=>"white", :transparency=>0.0,
:height=>0.05, :z_coord=>0.05, :id=>14, :radius=>10.0, :x_coord=>25.0}}}
```

```

{:command=>"add", :parameters=>

  {:y_coord=>10.0, :shape=>"cylinder", :colour=>"green", :transparency=>0.0,

   :height=>0.05, :z_coord=>0.1, :id=>15, :radius=>9.5, :x_coord=>25.0}},

{:command=>"add", :parameters=>

  {:y_coord=>10.0, :shape=>"cylinder", :colour=>"white", :transparency=>0.0,

   :height=>0.05, :z_coord=>0.25, :id=>16, :radius=>0.5, :x_coord=>25.0}},

{:command=>"add", :parameters=>

  {:y_coord=>12.0, :shape=>"cone", :colour=>"red", :transparency=>0.0,

   :height=>3.0, :z_coord=>0.0, :id=>40, :radius=>1.0, :x_coord=>30.0}},

{:command=>"add", :parameters=>

  {:y_coord=>12.0, :shape=>"cylinder", :colour=>"red", :transparency=>0.5,

   :height=>0.5, :z_coord=>0.0, :id=>41, :radius=>5.0, :x_coord=>30.0}},

{:command=>"add", :parameters=>

  {:y_coord=>15.0, :shape=>"cylinder", :colour=>"red", :transparency=>0.5,

   :height=>0.5, :z_coord=>0.0, :id=>73, :radius=>5.0, :x_coord=>25.0}},

{:command=>"add", :parameters=>

  {:y_coord=>13.0, :shape=>"cone", :colour=>"turquoise", :transparency=>0.0,

   :height=>3.0, :z_coord=>0.0, :id=>76, :radius=>1.0, :x_coord=>27.0}},

{:command=>"add", :parameters=>

  {:y_coord=>13.0, :shape=>"cylinder", :colour=>"red", :transparency=>0.5,

   :height=>0.5, :z_coord=>0.0, :id=>77, :radius=>5.0, :x_coord=>27.0}}]

```

## 7.4 Evaluating The Domain Objectives

Section 6.1.1 introduced the objectives of a domain specific language to represent interests. This section will evaluate Wish in terms of these objectives. Section 7.4.1 will consider abstraction, Section 7.4.2 readability, Section 7.4.3 succinctness, and finally Section 7.4.4 will consider the expressiveness of Wish.

Table 7.14: Artefacts within the Home Penalty Circle

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
10	box	30	0.05	15	25	7.5	0.15	NULL	white	0	pitch	0	home goal outer
11	box	29	0.05	14	25	7	0.2	NULL	green	0	pitch	0	home goal
14	cylinder	NULL	0.05	NULL	25	10	0.05	10	white	0	pitch	0	home penalty circle outer
15	cylinder	NULL	0.05	NULL	25	10	0.1	9.5	green	0	pitch	0	home penalty circle
16	cylinder	NULL	0.05	NULL	25	10	0.25	0.5	white	0	pitch	0	home penalty spot
40	cone	NULL	3	NULL	30	12	0	1	red	0	player	0	jim
41	cylinder	NULL	0.5	NULL	30	12	0	5	red	1	aura	0.5	jim
72	cone	NULL	3	NULL	25	15	0	1	blue	0	player	0	chris
73	cylinder	NULL	0.5	NULL	25	15	0	5	red	1	aura	0.5	chris
76	cone	NULL	3	NULL	27	13	0	1	turquoise	0	player	0	ref
77	cylinder	NULL	0.5	NULL	27	13	0	5	red	1	aura	0.5	ref

(x = x coord, y = y coord, z = z coord, v = virtual, t = transparency)

### 7.4.1 Abstraction

As explained in Section 6.2.5, Wish provides the concept of subwishes as an abstraction mechanism. Wish statements therefore allow the inclusion of other Wish statements to an arbitrary level of nesting as seen in Section 7.2.4. This facilitates the layering of abstractions, allowing complex statements to be broken into smaller, more manageable, components. These components can also aid the readability of Wish statements as discussed later in Sections 7.4.2.1 and 7.4.2.3. Subwishes can also be used within the scoping mechanism as described in Section 6.2.6. The same abstraction technique can therefore be used in two different ways depending on the context:

1. As a more readable shortcut for a nested set of Wish snippets (subwishes).
2. As a shortcut for a set of artefacts (scoping).

### 7.4.2 Readability

#### 7.4.2.1 Subwish Names

As described in Section 7.4.1 Wish's abstraction mechanism allows complex statements to be broken into smaller, more manageable components. The implementation of the subwish abstraction mechanism requires those smaller components to have unique names. If appropriate names are chosen, then the readability of the Wish statement can be improved. As an example of this, consider the following Wish condition:

```
name = sam
```

It is possible wrap this concept into a subwish as follows:

```
#named.wish
```

```
name = |A|
```



Now, if we use this subwish we can write the following, which is more natural and readable than the original condition:

```
named sam
```

#### 7.4.2.2 Visual Structure

Consider the following English interest statement:

*I am interested in the football pitch and all non-virtual artefacts that are red players, whose aura overlaps the referee's aura and that are within the home penalty circle*

The actual intention of the above statement may not be immediately visible from first reading. This is also the case with the equivalent SQL:

```
select * from artefacts where (category = 'pitch' or (not (virtual = true)) and ((colour = 'red') and
(category = 'player' and ((virtual = false and (category = 'player' and (name in
(select name from artefacts where (((5.0 + radius > sqrt(pow((x_coord - 27.0), 2) +
pow((y_coord - 13.0), 2))) and (category = 'aura')))))))) and ((9.5 > sqrt(pow((x_coord - 25.0), 2) +
pow((y_coord - 10.0), 2))))))))
```

However, consider the following Wish equivalent:

```
categorised_as pitch
not virtual
coloured red
categorised_as player
in_awareness_range_of ref_aura
within_circle home_penalty_circle
```

The Wish statement is clearly shorter and more succinct than the SQL version. It is also more visually organised than the English version. Once a reader has become accustomed to the Wish structure, the white-space forced by the syntax helps the parsing of the statement.

#### 7.4.2.3 Abstracting Complexity

As explained in Section 7.4.1, the Wish subwish mechanism allows for arbitrarily large, and arbitrarily nested Wish snippets to be abstracted. Not only can the choice of subwish name increase readability (as described in Section 7.4.2.1) the fact that the complexity is hidden means that there is less to read, therefore making the statement more readable. For an example of this concept consider the following subwish:

```
#within_box.wish
x_coord >= <%= |A|.x_coord - (|A|.width / 2)%>
x_coord <= <%= |A|.x_coord + (|A|.width / 2)%>
y_coord >= <%= |A|.y_coord - (|A|.length / 2)%>
y_coord <= <%= |A|.y_coord + (|A|.length / 2)%>
id != <%= |A|.id%>
```

#### 7.4.2.4 Removing Ambiguity

Consider the following English statement:

*I am interested in artefacts named sam and artefacts coloured red*

Note that the above statement is *not* the same as the following statement:

*I am interested in artefacts named sam and coloured red*

Both sentences use the word *and* to combine the conditions, yet they have very different meanings.

In SQL you would represent the first statement as follows:

```
name = 'sam' or colour = 'red'
```

And the second statement as follows:

```
name = 'sam' and colour = 'red'
```

Clearly there is an ambiguity between the English use of *and*, and SQL's use of the term. Wish removes this ambiguity by removing the *and* and *or* keywords entirely. Wish assumes that the statement is a simple list of the kind of artefacts that are interesting. In this case we're interested in artefacts named sam, and artefacts coloured red:

```
named sam
coloured red
```

If, however, we are interested in red artefacts named sam, Wish uses indentation to indicate that the second condition also applies to the first as follows:

```
named sam
  coloured red
```

### 7.4.3 Succinctness

Wish is syntactically and conceptually more succinct than SQL for representing interest statements. Wish supports abstraction for converting potentially complex snippets into single conditions. The scoping mechanism allows the representation of derived sets for use in interest conditions, and the removal of the *and* and *or* SQL keywords also makes the resulting statement more succinct.

### 7.4.4 Expressiveness

As Section 5.3 illustrated, SQL is sufficiently expressive for representing interest statements.<sup>6</sup>

Sections 5.3.1, 5.3.2, 5.3.3, and 5.3.4 give examples of the expressiveness of SQL, showing that it is capable of handling all of the interest statements types expressed by the examples in Section 3.3. Therefore the design decision to base Wish on top of SQL does not limit the potential expressiveness of Wish.

Section 6.1.1.4 described the requirement for expressiveness to be the ability to represent the examples presented in Section 3.3. As Section 7.2 illustrated, Wish is able to represent all these examples. This demonstrates that Wish is sufficiently expressive for the domain it is intended for: namely the representation of interest statements for virtual environments.

---

<sup>6</sup>However, it must be noted that this does not necessarily demonstrate that the use SQL is practical for efficiently implementing the processes that will enact these interests. For example, calculating line of sight requires the use of an auxiliary predicate function, which, under certain circumstances might be required to be executed for all pairs of artefacts to fully determine the visibility status of the system. For very large environments, this operation is clearly very costly, and could very easily exceed the processing capacity of the system.

## Chapter 8

# Conclusions and Further Work

Chapter 5 described the objectives and goals of this thesis. This chapter aims to summarise if, and how, these goals were met. It will then follow with an inspection of the potential new directions that could be taken by future work in this area. Section 8.1 provides summaries of the research contributions, and the chapters in this thesis. Section 8.2 discusses some final thoughts about this research in general, and finally Chapter 8.3 introduces a few potential directions that new work based on this research may take.

### 8.1 Summaries

#### 8.1.1 Contribution Summary

Section 1.7 listed a number of the contributions made to the research within the field of interest management for virtual environments. This section will revisit these contributions, and describe where this thesis also introduced them.

**Taxonomy of currently used interest management techniques.** The various techniques used for interest management were surveyed and discussed in Section 2.2.3. Section 2.2.4 introduced categorisation, locales and interacting locales as three general techniques, and Section 2.2.4.5 described various surveyed techniques that could be mapped on to them.

**A definition and conceptual model of virtual environments.** The term virtual environment was evaluated and discussed in Section 2.1.1. Section 2.1.1.1 illustrated that there is no general consensus on a definition, and a new definition was introduced in Section 2.1.1.3. Section 3.1.2 translated this definition into a number of axioms which were implemented in Chapter 4 as a proof of concept.

**A conceptual model of interests based on set-theory.** The taxonomy of interest management techniques presented in Section 2.2 was formalised using set-theory in Section 3.2, and then implemented using SQL as a proof of concept in Chapter 5.

**Wish, a domain specific language for representing interests.** Section 5.5 illustrated a number of

limitations in the implementation of the formalisation of the interest management techniques provided in Chapter 5. These limitations were shown to be a lack of readability, succinctness and no ability to allow for abstractions. These limitations are overcome through the inception of a new domain specific language. This new language, Wish, was introduced in Chapter 6 and then evaluated using a case study in Chapter 7.

### 8.1.2 Chapter Summary

This section provides a summary of each of the chapters found within this thesis.

1. **Introduction** This chapter introduced the general context of this thesis: namely issues of managing interests within virtual environments. It described scalability and adaptability as two limitations of current virtual environment implementations, and introduced dynamic interest management as a technique that could reduce the impact of these limitations.
2. **Literature Survey** This chapter surveyed the research literature on interest management within virtual environments. Based on this survey, it introduced definitions of the terms *interest management* and *virtual environment*. The chapter also described the various techniques used to manage interests, and showed how they can be mapped on to the following three general techniques: categorisation, locales, interacting locales.
3. **A Framework for Interest Management** This chapter introduced a set of axioms describing the fundamentals of virtual environments. Based on the constraints of these axioms, it then developed a formalisation of the three general interest management techniques introduced in the literature survey.
4. **Virtual Environment Axioms: A Proof of Concept** This chapter introduced a proof of concept of the virtual environment axioms by implementing them. The various design decisions for the implementation were discussed, and a simple environment illustrated.
5. **Interest Statements** This chapter introduced a proof of concept of the formal framework of interest management by presenting an implementation using SQL. The chapter then described how this implementation could represent the three general interest management techniques introduced in the literature survey. Finally some of the limitations of this implementation were described: namely a lack of readability, succinctness and no ability to allow for abstractions.
6. **Wish: a DSL for Interest Statements** This chapter introduced an alternative implementation of the formal framework of interest management using a new bespoke language called Wish. The design motivations of this new language were shown to be the ability to represent the three general interest management techniques, whilst being readable, succinct, and able to represent abstractions.

- 7. Case Study and Evaluation** This chapter evaluated Wish through the use of a case study. The case study showed the various abilities of Wish whilst illustrating that this new language is sufficiently expressive, more readable, and succinct than the SQL implementation, whilst also providing the ability to represent abstractions.
- 8. Conclusions and Further Work** This chapter provided a summary of the thesis, and discussed new directions that future work could take<sup>1</sup>.

## 8.2 Final Thoughts

### 8.2.1 Technology Choices

The implementation of the ideas presented in this thesis was achieved through the use of mainly non-standard technologies<sup>2</sup>. However, on reflection, it turns out that the technology choices were remarkably appropriate to the work, having great impact on the productivity and the flexibility of the research. As mentioned in Section 4.3.3, Java was initially chosen as the main implementation language simply because it was the main language taught by the school, and the language used by the majority of the researchers. It was the obvious choice. However, I believe that it was not the best choice. Research is often a very agile practice, where new ideas are constantly conjured up. It makes a lot of sense to work with material that allows you to convert ideas to prototypes as quickly as possible, to not pay a large price for trying out new directions. For example, the design of the Wish syntax was such a practice. The syntax constantly changed, and evolved from the SQL it produces to the syntax presented in Chapter 6. That evolution took a great many iterations, yet each iteration was a lot less work and effort using Ruby as the implementation language than it would have taken using Java. This is not to say that Ruby is a better language than Java, just one that is more suited to the task of rapid prototyping, and agile development. Ruby also provides a lot of features that would be almost impossibly hard to emulate with Java - for example it allowed the separation of the declaration of the relative artefacts and the Wish statements over separate files. The use of the Ruby on Rails framework was also an interesting choice - given that its intended use is for developing web applications. However, the Rails framework offered a sensible skeleton structure for the implementation, and an integrated console for interactive development. It also included, by default, all of the libraries that I intended to use, such as Active Record. This all meant that I could start developing my implementation much sooner than had I had to design and create my development context myself. I believe that making mistakes is an important part of the research process. I therefore believe that it is sensible to use technologies that are forgiving for such mistakes, and allow you to move on and try something different until you get it right.

<sup>1</sup>Well, actually this discussion is yet to come, yet it was already written whilst writing this chapter summary. Oh, the confusion of tenses!

<sup>2</sup>Where by non-standard I mean non-Java and non-Microsoft technologies.

### 8.2.2 Treating Wish as an Essay

Yukihiro Matsumoto, the creator of Ruby, wrote an article entitled *Treating Code as an Essay*[79]. The main thesis of this article was that he believed that like an essay, code should not just have a message, but that the message should also be easily understandable to humans. Therefore, structure, style and syntax play an important part of making the message of the code more readily digestible. He introduced the following example of how reducing syntactic clutter can make code more readable. Consider the following Rake<sup>3</sup> snippet:

```
task :default => [:test]

task :test do

  ruby "test/unittest.rb"

end
```

This code enjoys a lack of syntax often enforced by other languages such as missing method parameters, unbraced hash key/value pairs, and the ability to attach code block to the end of method calls. The same code written with this syntax included is as follows:

```
task({:default => [:test]})

task(:test, &lambda(){

  ruby "test/unittest.rb"

})
```

This philosophy is clearly inline with that which motivated the design of Wish. Wish removes syntax from SQL statements such as string quotations, parenthesis, and and or operators and explicit subqueries. As an example of this consider the following two equivalent statements taken from the case study in Chapter 7:

**Wish:**

```
not virtual

coloured red

categorised_as player

in_awareness_range_of ref_aura

within_circle home_penalty_circle
```

**SQL:**

```
select * from artefacts where (virtual = false and ((colour = 'red') and

(category = 'player' and ((virtual = false and (category = 'player' and
```

---

<sup>3</sup>Rake is a build tool similar to UNIX make, and Java ant.

```
(name in (select name from artefacts where
  (((5.0 + radius > sqrt(pow((x_coord - 27.0), 2) + pow((y_coord - 13.0), 2)))
  and (category = 'aura')))))) and
  ((9.5 > sqrt(pow((x_coord - 25.0), 2) + pow((y_coord - 10.0), 2))))))
```

Like an essay, Wish is intended to be readable by humans, therefore allowing it to be maintained by humans. I believe that Wish achieves this.

### 8.2.3 Component Objectives

Section 6.3.3 described that Wish consists of the following main components (illustrated in Figure 6.1):

- Ruby
  - Expressions
  - Relative Artefacts
- Wish
  - Explicit `not` and `in` operators
  - Explicit Groupings
  - Implicit Groupings
  - Abstraction
  - Autoquoting
  - Use of Abstraction for Scoping
- YAML with embedded SQL
  - Relative Interest Conditions
  - Scoping
  - Interest Conditions
  - Grouping
  - Conditional Operators

Each of the above components was introduced into the design of Wish for one of three reasons: Expressiveness, succinctness or readability. Figure 8.1 provides a spatial illustration of the motivations for the components in terms of these three reasons. From this diagram we can infer that the Wish and Ruby components provided much of the expressiveness of the language. This makes sense as the pure



SQL implementation provided in Chapter 5 was demonstrated to be expressive enough for the needs of the domain. We can also see the Wish components that were explicitly introduced for the motivations of readability and succinctness, and some components that were introduced for expressiveness in order to deal with the constraints of the Wish syntax.

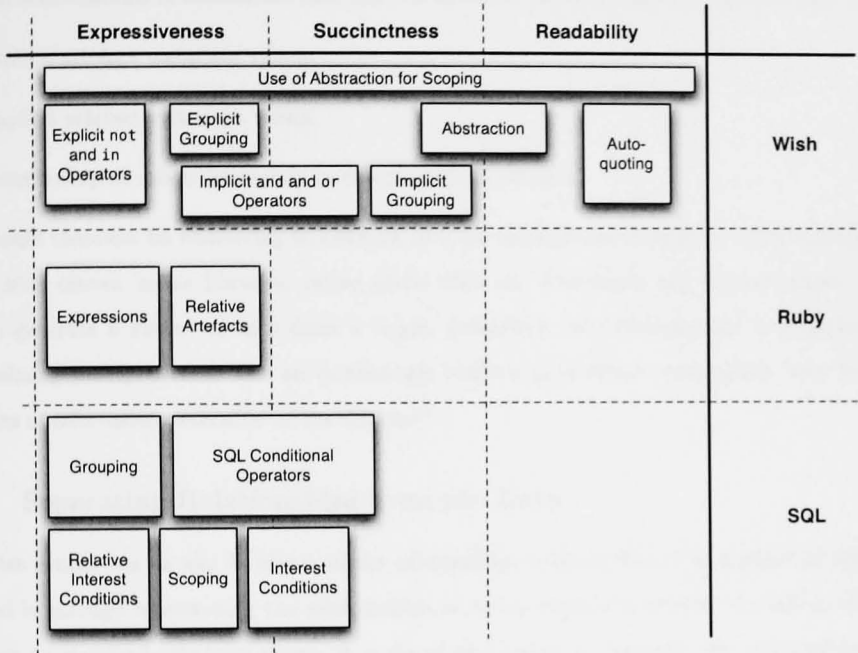


Figure 8.1: Spatial Map Indicating the Objectives of Wish Components

### 8.3 Further Work

Section 2.2.3 described the range of interest management techniques found within the literature. Each of these techniques is a method for expressing interests. It is probably fair to say that expressing interests has been a research topic from very early on in the history of virtual environments. However, although this thesis has been concerned with the expression of interests, its main focus has been on the representation of interests. Chapter 3 introduced a representation with set theory, Chapter 5 introduced a representation using SQL, and Chapter 6 introduced a representation using a new language called Wish - a language specific to the domain of expressing interests. Although I believe that there is still a lot of interesting research in the study of expressing interests, I believe that the study of representing interests has only started, and is a fertile research topic, even out of the context of virtual environments. This section will introduce some directions that such new research could take.

### 8.3.1 Wish as a DSL for Information Scoping

I believe that although the work presented in this thesis was conceived and evaluated within the context of interest management within virtual environments, it has the potential to be useful in a variety of other contexts. The Wish language is essentially a DSL for scoping information from a large set to a smaller set based on combinations of statements that can reason about the following information where provided:

- explicit artefact metadata values,
- implicit artefact metadata values,
- membership of values (metadata or other) in given subsets.

It would therefore be interesting to research into the applications of Wish in other domains such as emails, blog entries, music libraries, online photo sites, etc. Essentially any context where you might wish to generate a subset of data from a larger, potentially overwhelming set. I mention the term overwhelming purely because this an increasingly realistic proposition, particularly with increasingly large sets of information available on the internet<sup>4</sup>.

### 8.3.2 Separating Relationships from the Data

The Wish syntax has no way to reason about relationships between data. The method by which this is achieved is through representing the relationships as either explicit or implicit metadata, which Wish can treat as standard attribute values. It is therefore possible to represent the relationships between artefacts in an external file - such as the relative artefacts file<sup>5</sup>. This is true in the case study, where the relationship between the referee and his/her aura is explicitly defined. It would be equally possible to represent these relationships as methods in the objects that represent the artefacts. So, instead of having two variables: `ref` and `ref.aura`, it would be possible to call the relationship from the referee object as follows: `ref.aura`.

It would therefore be interesting to see how this separation deals with a variety of potentially complex case studies, and how the structure of those relationships might be represented. It would also be interesting to study the ability to have a variety of different relationship declarations for the same set of artefacts, and consider contexts where altering relationship types could be useful.

It would also be interesting to determine the performance characteristics of different types of relationships, such as visibility, defined with a number of different algorithms. Section 6.2.2 identified that certain algorithms could potentially represent a performance hit, particularly within hugely crowded environments. A study could be made into the type of algorithms that induce such a performance hit, and possible methods for increasing the efficiency of determining such relationships.

<sup>4</sup>For example, from social web applications such as flickr[123], last.fm[71] and del.icio.us[122].

<sup>5</sup>This would be similar to how style is separated from content on the web today. This separation happens typically with different files a `html` file for the content, and a `css` file for the style.

### 8.3.3 Resource Costs

This Thesis has attempted to ignore the practicality of any potential resource costs caused by the techniques introduced. This has been necessary in order to focus on the representations of interest rather than the processes necessary to enact those representations. Clearly the process of enacting the representations is important if the representations are to be used in real-world applications.

One of the main motivations for for designing Wish was the relative complexity of the SQL necessary to represent simple interest statements. For more complex statements, whilst Wish has the ability to abstract away from the complexity, it is still important to note that the relative complexity of the corresponding SQL may lead to excessive SQL queries needing to be executed in order to interpret the interest statement and generate the set of interesting artefacts.

It would therefore be very important to study the resource costs of employing these techniques to determine the performance characteristics and effect on the scalability of complex systems, and their ability to implement a variety of interest statements (such as those described in the case study) with really large and crowded environments.

### 8.3.4 Rich Interest Conditions

Section 5.5.1 introduced the limitations of SQL in representing rich interest conditions such as visibility. It be very interesting to attempt to implement some of these conditions and incorporate them into the design of Wish. It also be interesting to examine the performance implications of such conditions on the system as a whole.

### 8.3.5 Compiling Wish to Other Representations

Currently the Wish compiler converts Wish statements to SQL. It would be interesting to see what other output formats would be useful for the Wish compiler. Consider the following cases:

**Object Persistence Libraries** Many object-oriented languages have object persistence libraries - essentially storing sets of objects, or artefacts. Wish could be used to scope sets of such artefacts.

**XPath Queries** Increasingly large amounts of data are now stored in XML documents, or even XML databases. It would be interesting to evaluate the usefulness of Wish to scope sets of XML artefacts.

**Web APIs** Given the increasing number of interfaces to web services offering huge sets of data, it would be interesting to evaluate the ability for Wish to interact with such services. For example, a Wish statement could be sent inside a simple HTTP packet to a REST web service, or nested within an XML document within a SOAP envelope within an HTTP packet for a WS-\* service<sup>6</sup>.

---

<sup>6</sup>Surely a web service should mimic the web, rather than abstract it away.

**CouchDB** CouchDb[63] is a document-oriented, non-relational database management server. It stores sets of name-value pairs and associated metadata as documents. This design is clearly similar to the approach taken by the implementation of this thesis. It would therefore be interesting to investigate an implementation of Wish using this platform.

### 8.3.6 Interesting Events

This thesis has focussed on representations of expressions regarding sets of artefacts. In particular, the set of interesting artefacts. However, given that one of the main motivations for interest management is scalability, and that this motivation is tightly coupled with the number of messages that the system needs to send, it would be interesting to look at interest statements that reason about events instead of artefacts. It might also be possible to look at interest statements that are able to express an interest in both events and artefacts. Consider the following hypothetical event-oriented wish statement:

```
#I'm interested in events that match the following

category = important
size < <%= MY_MAX_SIZE%>
affects interesting_artefacts
```

In the above statement, both category and size are both message attributes. affects is a new keyword which is similar to in, where it refers to messages that affect any of the artefacts in the set represented by the standard subwish interesting.artefacts.

### 8.3.7 Prioritised Events

This thesis has focussed on representing sets - sets of interesting artefacts, sets of enforced artefacts, sets of all world artefacts, etc. Sets themselves have no ordering, they are just bags of information. It might be useful to not only extract a set of interesting artefacts, but add priorities and other metadata. It might therefore be possible to create statements such as the following:

*"I'm more interested in artefacts closer to me than artefacts further away"*

## Appendix A

# Example Iteration

### A.1 Wish Auto-quoting Implementation

```
def auto_quote(term)

  #define some useful regexp matchers:

  backticks = /^`(.+)`$/
  quotes = /^'.+'$/
  numeric = /^[-+]?[0-9]*\.[0-9]+$/
  true_or_false = /^(true|false)$/
  tagged_expression = /<%=.*%>/
  untagged_expression = /^[\^.\S*\.\S*[\^.]$/

  #add appropriate quotation tags if necessary:

  case term

  when backticks

    #check to see if backticks need to be removed

    return @remove_backticks ? (term.match backticks)[1] : term

  when quotes, numeric, true_or_false, tagged_expression

    return term

  when untagged_expression

    return "<%= #{term}%>"

  else

    return "'#{term}'"

  end

end
```

## A.2 Auto-quoting Specification Output

wish pre-parser auto-quoting

- should ignore a single digit
- should ignore many digits
- should ignore a negative digit
- should ignore a decimal
- should ignore a negative decimal
- should ignore a quoted integer
- should quote the word four
- should quote a word ending in digits
- should quote a word starting in digits
- should ignore the boolean value true
- should ignore the boolean value false
- should quote the word truth
- should put expression tags around a word containing a .
- should put expression tags around a word containing multiple .'s
- should ignore something that looks like expression but is surrounded by quotes
- should ignore explicit expression tags
- should quote a word that ends with a period (but does not contain any)
- should quote a word that ends with a period (and even contains one)
- should quote a word that starts with a period (and even contains one)
- should quote a word that starts and ends with a period (and even contains one)
- should quote a word that starts and ends with a period (and does not contain any)
- should put quote a word starting with a period
- should ignore a word that ends with a period, contains a space and is already quoted
- should quote a line even with spaces
- should ignore anything that is surrounded with backticks
- should quote anything that includes one or more spaces
- should not quote anything surrounded with backticks
- should quote anything that ends with multiple periods
- should quote anything that contains spaces, even if it contains a single period
- should quote anything that contains spaces, even if it contains multiple periods too
- should be able to ignore back-ticks, even when generated with an expression
- should not quote an expression and other terms, however should auto-quote the result

. should not quote an expression and other terms, however should auto-quote the result

Finished in 1.32405 seconds

34 examples, 0 failures

### A.3 Example RSpec Specification: Auto-quoting

```
describe "wish auto-quoting" do
```

```
  before(:each) do
```

```
    @interest = Interests.new(false, false, false)
```

```
  end
```

```
  it "should ignore a single digit" do
```

```
    wish = "attribute = 4"
```

```
    post_parse = "(attribute = 4)"
```

```
    @interest.parse_interests(wish, "").should == post_parse
```

```
  end
```

```
  it "should ignore many digits" do
```

```
    wish = "attribute = 12345678901234567890"
```

```
    post_parse = "(attribute = 12345678901234567890)"
```

```
    @interest.parse_interests(wish, "").should == post_parse
```

```
  end
```

```
  it "should ignore a negative digit" do
```

```
    wish = "attribute = -4"
```

```
    post_parse = "(attribute = -4)"
```

```
    @interest.parse_interests(wish, "").should == post_parse
```

```
  end
```

```
  it "should ignore a decimal" do
```

```
    wish = "attribute = 4.0"
```

```
    post_parse = "(attribute = 4.0)"
```

```
@interest.parse_interests(wish, "").should == post_parse
end
```

```
it "should ignore a negative decimal" do
  wish = "attribute = -4.0"
  post_parse = "(attribute = -4.0)"
  @interest.parse_interests(wish, "").should == post_parse
end
```

```
it "should ignore a quoted integer" do
  wish = "attribute = '4'"
  post_parse = "(attribute = '4')"
```

```
@interest.parse_interests(wish, "").should == post_parse
end
```

```
it "should quote the word four" do
  wish = "attribute = four"
  post_parse = "(attribute = 'four')"
```

```
@interest.parse_interests(wish, "").should == post_parse
end
```

```
it "should quote a word ending in digits" do
  wish = "attribute = cock-ver10"
  post_parse = "(attribute = 'cock-ver10')"
```

```
@interest.parse_interests(wish, "").should == post_parse
end
```

```
it "should quote a word starting in digits" do
  wish = "attribute = 1st"
  post_parse = "(attribute = '1st')"
```

```
@interest.parse_interests(wish, "").should == post_parse
end
```

```
it "should ignore the boolean value true" do
  wish = "attribute = true"
```



```

post_parse = "(attribute = true)"
@interest.parse_interests(wish, "").should == post_parse
end

```

```

it "should ignore the boolean value false" do
  wish = "attribute = false"
  post_parse = "(attribute = false)"
  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it "should quote the word truth" do
  wish = "attribute = truth"
  post_parse = "(attribute = 'truth')"
  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it "should put expression tags around a word containing a . " do
  wish = "attribute = 10.next"
  post_parse = "(attribute = 11)"
  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it "should put expression tags around a word containing multiple .'s " do
  wish = "attribute = 10.next.next"
  post_parse = "(attribute = 12)"
  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it "should ignore something that looks like expression but is surrounded by quotes" do
  wish = "attribute = 'x.value.sub_value'"
  post_parse = "(attribute = 'x.value.sub_value')"
  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it "should ignore explicit expression tags" do

```

```

wish = "attribute = <%= 10.next %>"
post_parse = "(attribute = 11)"
@interest.parse_interests(wish, "").should == post_parse
end

it "should quote a word that ends with a period (but does not contain any)" do
  wish = "attribute = end."
  post_parse = "(attribute = 'end.')"
  @interest.parse_interests(wish, "").should == post_parse
end

it "should quote a word that ends with a period (and even contains one)" do
  wish = "attribute = why.end."
  post_parse = "(attribute = 'why.end.')"
  @interest.parse_interests(wish, "").should == post_parse
end

it "should quote a word that starts with a period (and even contains one)" do
  wish = "attribute = .why.end"
  post_parse = "(attribute = '.why.end')"
  @interest.parse_interests(wish, "").should == post_parse
end

it "should quote a word that starts and ends with a period (and even contains one)" do
  wish = "attribute = .why.end."
  post_parse = "(attribute = '.why.end.')"
  @interest.parse_interests(wish, "").should == post_parse
end

it "should quote a word that starts and ends with a period (and does not contain any)" do
  wish = "attribute = .why."
  post_parse = "(attribute = '.why.')"
  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it "should put ignore a decimal with no preceding digit" do
  wish = "attribute = .5"
  post_parse = "(attribute = .5)"
  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it "should put quote a word starting with a period" do
  wish = "attribute = .rb"
  post_parse = "(attribute = '.rb'))"
  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it "should ignore a word that ends with a period, contains a space and is already quoted" do
  wish = "attribute = 'The End.'"
  post_parse = "(attribute = 'The End.')"
  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it "should quote a line even with spaces" do
  wish = "attribute = The End."
  post_parse = "(attribute = 'The End.')"
  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it 'should ignore anything that is surrounded with backticks' do
  wish = "attribute = `sqrt(81)`"
  post_parse = "(attribute = sqrt(81))"
  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it 'should quote anything that includes one or more spaces' do
  wish = "attribute = 2 + 4"
  post_parse = "(attribute = '2 + 4'))"
  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it 'should not quote anything surrounded with backticks' do
  wish = "attribute = `2 + 4`"
  post_parse = "(attribute = 2 + 4)"
  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it 'should quote anything that ends with multiple periods' do
  wish = "attribute = .one.more.thing...."
  post_parse = "(attribute = '.one.more.thing....')"
```

```

  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it 'should quote anything that contains spaces, even if it contains a single period' do
  wish = "attribute = Getting. Tired"
  post_parse = "(attribute = 'Getting. Tired')"
```

```

  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it 'should quote anything that contains spaces, even if it contains multiple periods too' do
  wish = "attribute =   Getting. Very. Tired"
  post_parse = "(attribute = 'Getting. Very. Tired')"
```

```

  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it 'should be able to ignore back-ticks, even when generated with an expression' do
  wish = "attribute = <%= `2 + 4`%>"
  post_parse = "(attribute = 2 + 4)"
  @interest.parse_interests(wish, "").should == post_parse
end

```

```

it 'should not quote an expression and other terms, however should auto-quote the result' do
  wish = "attribute = <%= `2 + 4`%> + 5"
  post_parse = "(attribute = '2 + 4 + 5')"
```

```

  @interest.parse_interests(wish, "").should == post_parse

```

end

it 'should not quote an expression and other terms, however should auto-quote the result' do

wish = "attribute = <%= '2'%>5"

post\_parse = "(attribute = 25)"

@interest.parse\_interests(wish, "").should == post\_parse

end

end

## Appendix B

# Case Study Data

### B.1 Football Pitch

Table B.1: Football Pitch Artefacts

ld	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
5	box	50	0.05	50	25	25	0	NULL	green	0	pitch	0	home half
6	box	50	0.05	50	25	75	0	NULL	green	0	pitch	0	away half
7	box	50	0.05	0.5	25	50	0.1	NULL	white	0	pitch	0	centre line
8	cylinder	NULL	0.05	NULL	25	50	0.05	7	white	0	pitch	0	centre circle outer
9	cylinder	NULL	0.05	NULL	25	50	0.1	6.5	green	0	pitch	0	centre circle
10	box	30	0.05	15	25	7.5	0.15	NULL	white	0	pitch	0	home goal outer
11	box	29	0.05	14	25	7	0.2	NULL	green	0	pitch	0	home goal
12	box	30	0.05	15	25	92.5	0.15	NULL	white	0	pitch	0	away goal outer
13	box	29	0.05	14	25	93	0.2	NULL	green	0	pitch	0	away goal
14	cylinder	NULL	0.05	NULL	25	10	0.05	10	white	0	pitch	0	home penalty circle outer
15	cylinder	NULL	0.05	NULL	25	10	0.1	9.5	green	0	pitch	0	home penalty circle
16	cylinder	NULL	0.05	NULL	25	10	0.25	0.5	white	0	pitch	0	home penalty spot
17	cylinder	NULL	0.05	NULL	25	90	0.05	10	white	0	pitch	0	away penalty circle outer
18	cylinder	NULL	0.05	NULL	25	90	0.1	9.5	green	0	pitch	0	away penalty circle
19	cylinder	NULL	0.05	NULL	25	90	0.25	0.5	white	0	pitch	0	away penalty spot
20	box	0.5	0.05	100	-0.25	50	0.1	NULL	white	0	pitch	0	near touch line
21	box	0.5	0.05	100	50.25	50	0.1	NULL	white	0	pitch	0	far touch line
22	box	51	0.05	0.5	25	-0.25	0.1	NULL	white	0	pitch	0	home touch line
23	box	51	0.05	0.5	25	100.25	0.1	NULL	white	0	pitch	0	away touch line
24	box	5	0.05	100	-2.5	50	0	NULL	green	0	pitch	0	near touch area
25	box	5	0.05	100	52.5	50	0	NULL	green	0	pitch	0	far touch area
26	box	60	0.05	5	25	-2.5	0	NULL	green	0	pitch	0	home touch area
27	box	60	0.05	5	25	102.5	0	NULL	green	0	pitch	0	away touch area
28	box	0.5	5	0.5	20	0	0	NULL	white	0	pitch	0	home near goal post
29	box	0.5	5	0.5	30	0	0	NULL	white	0	pitch	0	home far goal post
30	box	10.5	0.5	0.5	25	0	5	NULL	white	0	pitch	0	home goal crossbar
31	box	10.5	0.5	0.5	25	100	5	NULL	white	0	pitch	0	away goal crossbar
32	box	0.5	5	0.5	20	100	0	NULL	white	0	pitch	0	away near goal post
33	box	0.5	5	0.5	30	100	0	NULL	white	0	pitch	0	away far goal post

(x = x coord, y = y coord, z = z coord, v = virtual, t = transparency)

## B.2 Players

### B.2.1 Red Team

Table B.2: All Red Players

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
34	cone	NULL	3	NULL	30	48	0	1	red	0	player	0	sam
36	cone	NULL	3	NULL	50	38	0	1	red	0	player	0	bob
38	cone	NULL	3	NULL	10	8	0	1	red	0	player	0	john
40	cone	NULL	3	NULL	30	12	0	1	red	0	player	0	jim
42	cone	NULL	3	NULL	40	70	0	1	red	0	player	0	geoffrey
44	cone	NULL	3	NULL	40	90	0	1	red	0	player	0	bernard
46	cone	NULL	3	NULL	20	30	0	1	red	0	player	0	toddy
48	cone	NULL	3	NULL	40	43	0	1	red	0	player	0	charlie
50	cone	NULL	3	NULL	20	60	0	1	red	0	player	0	rupert
52	cone	NULL	3	NULL	5	20	0	1	red	0	player	0	sven

(x = x coord, y = y coord, z = z coord, v = virtual, t = transparency)

### B.2.2 Blue Team

Table B.3: All Blue Players

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
56	cone	NULL	3	NULL	30	60	0	1	blue	0	player	0	karl
58	cone	NULL	3	NULL	35	45	0	1	blue	0	player	0	hendrik
60	cone	NULL	3	NULL	15	13	0	1	blue	0	player	0	david
62	cone	NULL	3	NULL	45	85	0	1	blue	0	player	0	han
64	cone	NULL	3	NULL	35	14	0	1	blue	0	player	0	jean
66	cone	NULL	3	NULL	45	55	0	1	blue	0	player	0	boris
68	cone	NULL	3	NULL	35	95	0	1	blue	0	player	0	bilbo
70	cone	NULL	3	NULL	45	65	0	1	blue	0	player	0	savas
72	cone	NULL	3	NULL	25	15	0	1	blue	0	player	0	chris

(x = x coord, y = y coord, z = z coord, v = virtual, t = transparency)

### B.2.3 Goalkeepers

Table B.4: Goalkeepers

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
54	cone	NULL	3	NULL	26	0	0	1	yellow	0	goalie	0	tim
74	cone	NULL	3	NULL	25	100	0	1	yellow	0	goalie	0	carlos

(x = x coord, y = y coord, z = z coord, v = virtual, t = transparency)

## B.2.4 Referee

Table B.5: The Referee

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
76	cone	NULL	3	NULL	27	13	0	1	turquoise	0	player	0	ref

(*x = x coord, y = y coord, z = z coord, v = virtual, t = transparency*)

## B.3 Football

Table B.6: The Football

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
78	sphere	NULL	NULL	NULL	15	7	2	0.5	white	0	ball	0	ball

(*x = x coord, y = y coord, z = z coord, v = virtual, t = transparency*)

## B.4 Locales

Table B.7: Locales

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
79	box	25	0.05	100	12.5	50	1	NULL	brown	1	locale	0.5	near half

(*x = x coord, y = y coord, z = z coord, v = virtual, t = transparency*)

## B.5 Auras

## B.6 All Artefacts



Table B.8: Auras

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
35	cylinder	NULL	0.5	NULL	30	48	0	5	red	1	aura	0.5	sam
37	cylinder	NULL	0.5	NULL	50	38	0	5	red	1	aura	0.5	bob
39	cylinder	NULL	0.5	NULL	10	8	0	5	red	1	aura	0.5	john
41	cylinder	NULL	0.5	NULL	30	12	0	5	red	1	aura	0.5	jim
43	cylinder	NULL	0.5	NULL	40	70	0	5	red	1	aura	0.5	geoffrey
45	cylinder	NULL	0.5	NULL	40	90	0	5	red	1	aura	0.5	bernard
47	cylinder	NULL	0.5	NULL	20	30	0	5	red	1	aura	0.5	toddy
49	cylinder	NULL	0.5	NULL	40	43	0	5	red	1	aura	0.5	charlie
51	cylinder	NULL	0.5	NULL	20	60	0	5	red	1	aura	0.5	rupert
53	cylinder	NULL	0.5	NULL	5	20	0	5	red	1	aura	0.5	sven
55	cylinder	NULL	0.5	NULL	26	0	0	5	red	1	aura	0.5	tim
57	cylinder	NULL	0.5	NULL	30	60	0	5	red	1	aura	0.5	karl
59	cylinder	NULL	0.5	NULL	35	45	0	5	red	1	aura	0.5	hendrik
61	cylinder	NULL	0.5	NULL	15	13	0	5	red	1	aura	0.5	david
63	cylinder	NULL	0.5	NULL	45	85	0	5	red	1	aura	0.5	han
65	cylinder	NULL	0.5	NULL	35	14	0	5	red	1	aura	0.5	jean
67	cylinder	NULL	0.5	NULL	45	55	0	5	red	1	aura	0.5	boris
69	cylinder	NULL	0.5	NULL	35	95	0	5	red	1	aura	0.5	bilbo
71	cylinder	NULL	0.5	NULL	45	65	0	5	red	1	aura	0.5	savas
73	cylinder	NULL	0.5	NULL	25	15	0	5	red	1	aura	0.5	chris
75	cylinder	NULL	0.5	NULL	25	100	0	5	red	1	aura	0.5	carlos
77	cylinder	NULL	0.5	NULL	27	13	0	5	red	1	aura	0.5	ref

(x = x coord, y = y coord, z = z coord, v = virtual, t = transparency)

Table B.9: All Artefacts

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
5	box	50	0.05	50	25	25	0	NULL	green	0	pitch	0	home half
6	box	50	0.05	50	25	75	0	NULL	green	0	pitch	0	away half
7	box	50	0.05	0.5	25	50	0.1	NULL	white	0	pitch	0	centre line
8	cylinder	NULL	0.05	NULL	25	50	0.05	7	white	0	pitch	0	centre circle outer
9	cylinder	NULL	0.05	NULL	25	50	0.1	6.5	green	0	pitch	0	centre circle
10	box	30	0.05	15	25	7.5	0.15	NULL	white	0	pitch	0	home goal outer
11	box	29	0.05	14	25	7	0.2	NULL	green	0	pitch	0	home goal
12	box	30	0.05	15	25	92.5	0.15	NULL	white	0	pitch	0	away goal outer
13	box	29	0.05	14	25	93	0.2	NULL	green	0	pitch	0	away goal
14	cylinder	NULL	0.05	NULL	25	10	0.05	10	white	0	pitch	0	home penalty circle outer
15	cylinder	NULL	0.05	NULL	25	10	0.1	9.5	green	0	pitch	0	home penalty circle
16	cylinder	NULL	0.05	NULL	25	10	0.25	0.5	white	0	pitch	0	home penalty spot
17	cylinder	NULL	0.05	NULL	25	90	0.05	10	white	0	pitch	0	away penalty circle outer
18	cylinder	NULL	0.05	NULL	25	90	0.1	9.5	green	0	pitch	0	away penalty circle
19	cylinder	NULL	0.05	NULL	25	90	0.25	0.5	white	0	pitch	0	away penalty spot
20	box	0.5	0.05	100	-0.25	50	0.1	NULL	white	0	pitch	0	near touch line
21	box	0.5	0.05	100	50.25	50	0.1	NULL	white	0	pitch	0	far touch line
22	box	51	0.05	0.5	25	-0.25	0.1	NULL	white	0	pitch	0	home touch line
23	box	51	0.05	0.5	25	100.25	0.1	NULL	white	0	pitch	0	away touch line
24	box	5	0.05	100	-2.5	50	0	NULL	green	0	pitch	0	near touch area
25	box	5	0.05	100	52.5	50	0	NULL	green	0	pitch	0	far touch area
26	box	60	0.05	5	25	-2.5	0	NULL	green	0	pitch	0	home touch area
27	box	60	0.05	5	25	102.5	0	NULL	green	0	pitch	0	away touch area
28	box	0.5	5	0.5	20	0	0	NULL	white	0	pitch	0	home near goal post
29	box	0.5	5	0.5	30	0	0	NULL	white	0	pitch	0	home far goal post
30	box	10.5	0.5	0.5	25	0	5	NULL	white	0	pitch	0	home goal crossbar
31	box	10.5	0.5	0.5	25	100	5	NULL	white	0	pitch	0	away goal crossbar
32	box	0.5	5	0.5	20	100	0	NULL	white	0	pitch	0	away near goal post
33	box	0.5	5	0.5	30	100	0	NULL	white	0	pitch	0	away far goal post
34	cone	NULL	3	NULL	30	48	0	1	red	0	player	0	sam
35	cylinder	NULL	0.5	NULL	30	48	0	5	red	1	aura	0.5	sam
36	cone	NULL	3	NULL	50	38	0	1	red	0	player	0	bob
37	cylinder	NULL	0.5	NULL	50	38	0	5	red	1	aura	0.5	bob
38	cone	NULL	3	NULL	10	8	0	1	red	0	player	0	john
39	cylinder	NULL	0.5	NULL	10	8	0	5	red	1	aura	0.5	john
40	cone	NULL	3	NULL	30	12	0	1	red	0	player	0	jim
41	cylinder	NULL	0.5	NULL	30	12	0	5	red	1	aura	0.5	jim
42	cone	NULL	3	NULL	40	70	0	1	red	0	player	0	geoffrey
43	cylinder	NULL	0.5	NULL	40	70	0	5	red	1	aura	0.5	geoffrey
44	cone	NULL	3	NULL	40	90	0	1	red	0	player	0	bernard
45	cylinder	NULL	0.5	NULL	40	90	0	5	red	1	aura	0.5	bernard
46	cone	NULL	3	NULL	20	30	0	1	red	0	player	0	toddy
47	cylinder	NULL	0.5	NULL	20	30	0	5	red	1	aura	0.5	toddy
48	cone	NULL	3	NULL	40	43	0	1	red	0	player	0	charlie
49	cylinder	NULL	0.5	NULL	40	43	0	5	red	1	aura	0.5	charlie
50	cone	NULL	3	NULL	20	60	0	1	red	0	player	0	rupert

Continued in Table B.10

Table B.10: All Artefacts (Continued from Table B.9)

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
51	cylinder	NULL	0.5	NULL	20	60	0	5	red	1	aura	0.5	rupert
52	cone	NULL	3	NULL	5	20	0	1	red	0	player	0	sven
53	cylinder	NULL	0.5	NULL	5	20	0	5	red	1	aura	0.5	sven
54	cone	NULL	3	NULL	26	0	0	1	yellow	0	goalie	0	tim
55	cylinder	NULL	0.5	NULL	26	0	0	5	red	1	aura	0.5	tim
56	cone	NULL	3	NULL	30	60	0	1	blue	0	player	0	karl
57	cylinder	NULL	0.5	NULL	30	60	0	5	red	1	aura	0.5	karl
58	cone	NULL	3	NULL	35	45	0	1	blue	0	player	0	hendrik
59	cylinder	NULL	0.5	NULL	35	45	0	5	red	1	aura	0.5	hendrik
60	cone	NULL	3	NULL	15	13	0	1	blue	0	player	0	david
61	cylinder	NULL	0.5	NULL	15	13	0	5	red	1	aura	0.5	david
62	cone	NULL	3	NULL	45	85	0	1	blue	0	player	0	han
63	cylinder	NULL	0.5	NULL	45	85	0	5	red	1	aura	0.5	han
64	cone	NULL	3	NULL	35	14	0	1	blue	0	player	0	jean
65	cylinder	NULL	0.5	NULL	35	14	0	5	red	1	aura	0.5	jean
66	cone	NULL	3	NULL	45	55	0	1	blue	0	player	0	boris
67	cylinder	NULL	0.5	NULL	45	55	0	5	red	1	aura	0.5	boris
68	cone	NULL	3	NULL	35	95	0	1	blue	0	player	0	bilbo
69	cylinder	NULL	0.5	NULL	35	95	0	5	red	1	aura	0.5	bilbo
70	cone	NULL	3	NULL	45	65	0	1	blue	0	player	0	savas
71	cylinder	NULL	0.5	NULL	45	65	0	5	red	1	aura	0.5	savas
72	cone	NULL	3	NULL	25	15	0	1	blue	0	player	0	chris
73	cylinder	NULL	0.5	NULL	25	15	0	5	red	1	aura	0.5	chris
74	cone	NULL	3	NULL	25	100	0	1	yellow	0	goalie	0	carlos
75	cylinder	NULL	0.5	NULL	25	100	0	5	red	1	aura	0.5	carlos
76	cone	NULL	3	NULL	27	13	0	1	turquoise	0	player	0	ref
77	cylinder	NULL	0.5	NULL	27	13	0	5	red	1	aura	0.5	ref
78	sphere	NULL	NULL	NULL	15	7	2	0.5	white	0	ball	0	ball
79	box	25	0.05	100	12.5	50	1	NULL	brown	1	locale	0.5	near half

(x = x coord, y = y coord, z = z coord, v = virtual, t = transparency)

## Appendix C

# Case Study Example Statements

### C.1 Relative Artefacts

All the example statements in this chapter use have access to the subwishes defined in Appendix D, and use the following `relative_artefacts.rb` file:

### C.2 Categories

#### C.2.1 English Prose

*I am interested in all artefacts that are red*

#### C.2.2 Wish

`coloured red`

#### C.2.3 SQL

`select * from artefacts where ((colour = 'red'))`

#### C.2.4 Matching Artefacts

See Table C.1.

### C.3 Locales

#### C.3.1 English Prose

*I am interested in all artefacts within the near half of the pitch*

Table C.1: Red Artefacts

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
34	cone	NULL	3	NULL	30	48	0	1	red	0	player	0	sam
35	cylinder	NULL	0.5	NULL	30	48	0	5	red	1	aura	0.5	sam
36	cone	NULL	3	NULL	50	38	0	1	red	0	player	0	bob
37	cylinder	NULL	0.5	NULL	50	38	0	5	red	1	aura	0.5	bob
38	cone	NULL	3	NULL	10	8	0	1	red	0	player	0	john
39	cylinder	NULL	0.5	NULL	10	8	0	5	red	1	aura	0.5	john
40	cone	NULL	3	NULL	30	12	0	1	red	0	player	0	jim
41	cylinder	NULL	0.5	NULL	30	12	0	5	red	1	aura	0.5	jim
42	cone	NULL	3	NULL	40	70	0	1	red	0	player	0	geoffrey
43	cylinder	NULL	0.5	NULL	40	70	0	5	red	1	aura	0.5	geoffrey
44	cone	NULL	3	NULL	40	90	0	1	red	0	player	0	bernard
45	cylinder	NULL	0.5	NULL	40	90	0	5	red	1	aura	0.5	bernard
46	cone	NULL	3	NULL	20	30	0	1	red	0	player	0	toddy
47	cylinder	NULL	0.5	NULL	20	30	0	5	red	1	aura	0.5	toddy
48	cone	NULL	3	NULL	40	43	0	1	red	0	player	0	charlie
49	cylinder	NULL	0.5	NULL	40	43	0	5	red	1	aura	0.5	charlie
50	cone	NULL	3	NULL	20	60	0	1	red	0	player	0	rupert
51	cylinder	NULL	0.5	NULL	20	60	0	5	red	1	aura	0.5	rupert
52	cone	NULL	3	NULL	5	20	0	1	red	0	player	0	sven
53	cylinder	NULL	0.5	NULL	5	20	0	5	red	1	aura	0.5	sven
55	cylinder	NULL	0.5	NULL	26	0	0	5	red	1	aura	0.5	tim
57	cylinder	NULL	0.5	NULL	30	60	0	5	red	1	aura	0.5	karl
59	cylinder	NULL	0.5	NULL	35	45	0	5	red	1	aura	0.5	hendrik
61	cylinder	NULL	0.5	NULL	15	13	0	5	red	1	aura	0.5	david
63	cylinder	NULL	0.5	NULL	45	85	0	5	red	1	aura	0.5	han
65	cylinder	NULL	0.5	NULL	35	14	0	5	red	1	aura	0.5	jean
67	cylinder	NULL	0.5	NULL	45	55	0	5	red	1	aura	0.5	boris
69	cylinder	NULL	0.5	NULL	35	95	0	5	red	1	aura	0.5	bilbo
71	cylinder	NULL	0.5	NULL	45	65	0	5	red	1	aura	0.5	savas
73	cylinder	NULL	0.5	NULL	25	15	0	5	red	1	aura	0.5	chris
75	cylinder	NULL	0.5	NULL	25	100	0	5	red	1	aura	0.5	carlos
77	cylinder	NULL	0.5	NULL	27	13	0	5	red	1	aura	0.5	ref

(x = x coord, y = y coord, z = z coord, v = virtual, t = transparency)

### C.3.2 Wish

within\_box near\_half

### C.3.3 SQL

```
select * from artefacts where ((x_coord >= 0.0 and (x_coord <= 25.0 and (y_coord >= 0.0
and (y_coord <= 100.0 and (id != 79))))))
```

### C.3.4 Matching Artefacts

See Table C.2.

Table C.2: Artefacts on the Near Side of the Football Pitch

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
5	box	50	0.05	50	25	25	0	NULL	green	0	pitch	0	home half
6	box	50	0.05	50	25	75	0	NULL	green	0	pitch	0	away half
7	box	50	0.05	0.5	25	50	0.1	NULL	white	0	pitch	0	centre line
8	cylinder	NULL	0.05	NULL	25	50	0.05	7	white	0	pitch	0	centre circle outer
9	cylinder	NULL	0.05	NULL	25	50	0.1	6.5	green	0	pitch	0	centre circle
10	box	30	0.05	15	25	7.5	0.15	NULL	white	0	pitch	0	home goal outer
11	box	29	0.05	14	25	7	0.2	NULL	green	0	pitch	0	home goal
12	box	30	0.05	15	25	92.5	0.15	NULL	white	0	pitch	0	away goal outer
13	box	29	0.05	14	25	93	0.2	NULL	green	0	pitch	0	away goal
14	cylinder	NULL	0.05	NULL	25	10	0.05	10	white	0	pitch	0	home penalty circle outer
15	cylinder	NULL	0.05	NULL	25	10	0.1	9.5	green	0	pitch	0	home penalty circle
16	cylinder	NULL	0.05	NULL	25	10	0.25	0.5	white	0	pitch	0	home penalty spot
17	cylinder	NULL	0.05	NULL	25	90	0.05	10	white	0	pitch	0	away penalty circle outer
18	cylinder	NULL	0.05	NULL	25	90	0.1	9.5	green	0	pitch	0	away penalty circle
19	cylinder	NULL	0.05	NULL	25	90	0.25	0.5	white	0	pitch	0	away penalty spot
28	box	0.5	5	0.5	20	0	0	NULL	white	0	pitch	0	home near goal post
30	box	10.5	0.5	0.5	25	0	5	NULL	white	0	pitch	0	home goal crossbar
31	box	10.5	0.5	0.5	25	100	5	NULL	white	0	pitch	0	away goal crossbar
32	box	0.5	5	0.5	20	100	0	NULL	white	0	pitch	0	away near goal post
38	cone	NULL	3	NULL	10	8	0	1	red	0	player	0	john
39	cylinder	NULL	0.5	NULL	10	8	0	5	red	1	aura	0.5	john
46	cone	NULL	3	NULL	20	30	0	1	red	0	player	0	toddy
47	cylinder	NULL	0.5	NULL	20	30	0	5	red	1	aura	0.5	toddy
50	cone	NULL	3	NULL	20	60	0	1	red	0	player	0	rupert
51	cylinder	NULL	0.5	NULL	20	60	0	5	red	1	aura	0.5	rupert
52	cone	NULL	3	NULL	5	20	0	1	red	0	player	0	sven
53	cylinder	NULL	0.5	NULL	5	20	0	5	red	1	aura	0.5	sven
60	cone	NULL	3	NULL	15	13	0	1	blue	0	player	0	david
61	cylinder	NULL	0.5	NULL	15	13	0	5	red	1	aura	0.5	david
72	cone	NULL	3	NULL	25	15	0	1	blue	0	player	0	chris
73	cylinder	NULL	0.5	NULL	25	15	0	5	red	1	aura	0.5	chris
74	cone	NULL	3	NULL	25	100	0	1	yellow	0	goalie	0	carlos
75	cylinder	NULL	0.5	NULL	25	100	0	5	red	1	aura	0.5	carlos
78	sphere	NULL	NULL	NULL	15	7	2	0.5	white	0	ball	0	ball

(x = x coord, y = y coord, z = z coord, v = virtual, t = transparency)

## C.4 Relative Locales

### C.4.1 English Prose

*I am interested in all artefacts within the referee's aura*

### C.4.2 Wish

`within_circle ref_aura`

### C.4.3 SQL

```
select * from artefacts where ((5.0 > sqrt(pow((x_coord - 27.0), 2) +
pow((y_coord - 13.0), 2))))
```

### C.4.4 Matching Artefacts

See Table C.3.

Table C.3: Football Pitch Artefacts

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
14	cylinder	NULL	0.05	NULL	25	10	0.05	10	white	0	pitch	0	home penalty circle outer
15	cylinder	NULL	0.05	NULL	25	10	0.1	9.5	green	0	pitch	0	home penalty circle
16	cylinder	NULL	0.05	NULL	25	10	0.25	0.5	white	0	pitch	0	home penalty spot
40	cone	NULL	3	NULL	30	12	0	1	red	0	player	0	jim
41	cylinder	NULL	0.5	NULL	30	12	0	5	red	1	aura	0.5	jim
72	cone	NULL	3	NULL	25	15	0	1	blue	0	player	0	chris
73	cylinder	NULL	0.5	NULL	25	15	0	5	red	1	aura	0.5	chris
76	cone	NULL	3	NULL	27	13	0	1	turquoise	0	player	0	ref
77	cylinder	NULL	0.5	NULL	27	13	0	5	red	1	aura	0.5	ref

(*x* = *x* coord, *y* = *y* coord, *z* = *z* coord, *v* = *virtual*, *t* = *transparency*)

## C.5 Interacting Locales

### C.5.1 English Prose

*I am interested in all artefacts whose aura overlaps the referee's aura*

### C.5.2 Wish

`in_awareness_range_of ref_aura`

### C.5.3 SQL

```
select * from artefacts where ((virtual = false and (category = 'player' and (name in
(select name from artefacts where (((5.0 + radius > sqrt(pow((x_coord - 27.0), 2) +
pow((y_coord - 13.0), 2))) and (category = 'aura'))))))))
```

### C.5.4 Matching Artefacts

See Table C.4.

Table C.4: Artefacts Matching the Interacting Locales Example

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
40	cone	NULL	3	NULL	30	12	0	1	red	0	player	0	jim
64	cone	NULL	3	NULL	35	14	0	1	blue	0	player	0	jean
72	cone	NULL	3	NULL	25	15	0	1	blue	0	player	0	chris
76	cone	NULL	3	NULL	27	13	0	1	turquoise	0	player	0	ref

(x = x coord, y = y coord, z = z coord, v = virtual, t = transparency)

## C.6 Combinations

### C.6.1 English Prose

*I am interested in all non-virtual artefacts that are red players, whose aura overlaps the referee's aura and that are within the home penalty circle*

### C.6.2 Wish

```
not virtual
coloured red
categorised_as player
in_awareness_range_of ref_aura
within_circle home_penalty_circle
```

### C.6.3 SQL

```
select * from artefacts where (virtual = false and ((colour = 'red') and (category = 'player' and
((virtual = false and (category = 'player' and (name in (select name from artefacts where
(((5.0 + radius > sqrt(pow((x_coord - 27.0), 2) + pow((y_coord - 13.0), 2))) and (category = 'aura'))))))))
and ((9.5 > sqrt(pow((x_coord - 25.0), 2) + pow((y_coord - 10.0), 2))))))))
```



### C.6.4 Matching Artefacts

See Table C.5.

Table C.5: Artefacts Matching the Combinations Example

id	shape	width	height	length	x	y	z	radius	colour	v	category	t	name
40	cone	NULL	3	NULL	30	12	0	1	red	0	player	0	jim

(*x = x coord, y = y coord, z = z coord, v = virtual, t = transparency*)

# Appendix D

## Sub Wishes

### D.1 In Awareness Range Of

```
#in_awareness_range_of.wish

virtual = false

name in auras_in_awareness_range_of |A|
```

### D.2 Auras In awareness Range Of

```
#auras_in_awareness_range_of.wish

overlaps |A|

category = aura
```

### D.3 Overlaps

```
#overlaps.wish

<%=|A|.radius%> + radius > `sqrt(pow((x_coord - <%= |A|.x_coord%>), 2) + pow((y_coord - <%= |A|.y_coord%>), 2))`
```

### D.4 Coloured

```
#coloured.wish

colour = |A|
```

### D.5 Named

```
#named.wish

name = |A|
```

## D.6 Is

```
#is.wish
id = <%= |A|.id %>
```

## D.7 Within Circle

```
#within_circle.wish
<%=|A|.radius%> > `sqrt(pow((x_coord - <%= |A|.x_coord%>), 2) + pow((y_coord - <%= |A|.y_coord%>), 2))`
```

## D.8 Within Box

```
#within_box.wish
x_coord >= <%= |A|.x_coord - (|A|.width / 2)%>
x_coord <= <%= |A|.x_coord + (|A|.width / 2)%>
y_coord >= <%= |A|.y_coord - (|A|.length / 2)%>
y_coord <= <%= |A|.y_coord + (|A|.length / 2)%>
id != <%= |A|.id%>
```

## D.9 Within Cube

```
#within_cube.wish
x_coord >= <%= |A|.x_coord - (|A|.height / 2)%>
x_coord <= <%= |A|.x_coord + (|A|.height / 2)%>
y_coord >= <%= |A|.y_coord - (|A|.height / 2)%>
y_coord <= <%= |A|.y_coord + (|A|.height / 2)%>
id != <%= |A|.id%>
```

## D.10 Near To

```
#near_to.wish
x_coord >= <%= |A|.x_coord - 20 %>
x_coord <= <%= |A|.x_coord + 20 %>
y_coord >= <%= |A|.y_coord - 20 %>
y_coord <= <%= |A|.y_coord + 20 %>
id != <%= |A|.id%>
```

## D.11 Virtual

```
#virtual.wish  
virtual = true
```

## D.12 Categorised As

```
#categorised_as.wish  
category = |A|
```

# Bibliography

- [1] MySQL AB. Mysql. <http://www.mysql.com/>, August 2007.
- [2] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs - 2nd Edition (MIT Electrical Engineering and Computer Science)*. The MIT Press, September 1996.
- [3] Raffaele De Amicis, Giuseppe Conti, and Michele Fiorentino. Tangible interfaces in virtual environments for industrial design. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 261–264, New York, NY, USA, 2004. ACM Press.
- [4] Miguel Antunes, Antonio Rito Silva, and Jorge Martins. An abstraction for awareness management in collaborative virtual environments. In *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 33–39, New York, NY, USA, 2001. ACM Press.
- [5] R.B. Araujo, A. Boukerche, and N.J. McGraw. A grid-filtered region-based approach to support synchronization in large-scale distributed interactive virtual environments. *Parallel Processing, 2005. ICPP 2005 Workshops. International Conference Workshops on*, 2005.
- [6] Ken Arnold. Style is substance. In Joel Spolsky, editor, *The Best Software Writing*, volume I. Apress, 2005.
- [7] Dave Astels, Steven Baker, David Chelimsky, Aslak Hellesøy, and Brian Takita. Rspec. <http://rspec.rubyforge.org/>, August 2007.
- [8] Robert Bartlett. A categorisation model for distributed virtual environments. *IPDPS'04: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 231–, April 2004.
- [9] Mostafa A. Bassiouni, Ming-Hsing Chiu, Margaret Loper, Michael Garnsey, and Jim Williams. Performance and reliability analysis of relevance filtering for scalable distributed interactive simulation. *ACM Trans. Model. Comput. Simul.*, 7(3):293–331, 1997.
- [10] Kent Beck. Simple smalltalk testing: With patterns. Technical report, First Class Software, Inc., 1994.

- [11] Kent Beck. *Test-Driven Development By Example*. The Addison-Wesley Signature Series. Addison-Wesley, 2003.
- [12] Steve Benford and Lennart Fahlén. A spatial model of interaction in large virtual environments. In *ECSCW'93: Proceedings of the third conference on European Conference on Computer-Supported Cooperative Work*, pages 109–124, Norwell, MA, USA, 1993. Kluwer Academic Publishers.
- [13] Steve Benford, Chris Greenhalgh, and David Lloyd. Crowded collaborative virtual environments. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 59–66, New York, NY, USA, 1997. ACM Press.
- [14] Hrvoje Benko, Edward W. Ishak, and Steven Feiner. Collaborative mixed reality visualization of an archaeological excavation. *ismar*, 00:132–140, 2004.
- [15] Ashwin Bhambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: a distributed architecture for online multiplayer games. In *NSDI'06: Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design & Implementation*, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.
- [16] Gary Bishop and Henry Fuchs. Research directions in virtual environments — report of an NSF invitational workshop, March 23–24, 1992. *Computer Graphics*, 26(3):153–177, 1992.
- [17] A. Boukerche, N. J. McGraw, and R. B. Araujo. A novel data distribution management scheme to support synchronization in large-scale distributed virtual environments. In *Virtual Environments, Human-Computer Interfaces and Measurement Systems, 2005. VECIMS 2005. Proceedings of the 2005 IEEE International Conference on*, pages 6 pp.–, 2005.
- [18] W. Broll. Interacting in distributed collaborative virtual environments. In *VRAIS'95 - Virtual Reality Annual International Symposium*, pages 148–155, 1995.
- [19] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, 1995.
- [20] R. Brunton, P. McAndrews, K.L. Morse, J. Muguira, J.M. Pullen, and A. Tolk. An architecture for web-services based interest management in real ... *Distributed Simulation and Real-Time Applications, 2004. DS-RT 2004. Eighth IEEE International Symposium on*, 2004.
- [21] J. Calvin, D. Cebula, C. Chiang, S. Rak, and D. Van Hook. Data subscription in support of multicast group allocation. In *13th Workshop on Standards for the Interoperability of Distributed Simulations*, pages 367–369, September 1995.

- [22] J. Calvin, A. Dickens, B. Gaines, P. Metzger, D. Miller, and D. Owen. The SIMNET virtual world architecture. *Virtual Reality Annual International Symposium, 1993., 1993 IEEE*, pages 450–455, 1993.
- [23] Toga K. Capin and Daniel Thalmann. A taxonomy of networked virtual environments. In *IWSNHC3DI'99: International Workshop on Synthetic - Natural Hybrid Coding and Three Dimensional Imaging*, 1999.
- [24] Jin Chen, Baohua Wu, Margaret Delap, Bjorn Knutsson, Honghui Lu, and Cristiana Amza. Locality aware dynamic load management for massively multiplayer games. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 289–300, New York, NY, USA, 2005. ACM Press.
- [25] E.F. Churchill and D. Snowdon. Collaborative virtual environments: An introductory review of issues and systems. *Virtual Reality*, 3(1):3–15, 1998.
- [26] Elizabeth F. Churchill and Sara Bly. Virtual environments at work: ongoing use of muds in the workplace. *SIGSOFT Softw. Eng. Notes*, 24(2):99–108, 1999.
- [27] The Ruby Community. The ruby programming language. <http://ruby-lang.org/>, August 2007.
- [28] Monte Cook. *D&D Special Edition Dungeon Master's Guide (Dungeon & Dragons Roleplaying Game: Adventures)*. Wizards of the Coast, 2005.
- [29] Pavel Curtis. Mudding: Social phenomena in text-based virtual realities. In *Proceedings of the 1992 Conference on the Directions and Implications of Advanced Computing*, Berkeley, CA, 1992.
- [30] Dan North Dave Astels. Behaviour-driven development. <http://behaviour-driven.org/>, August 2007.
- [31] Jauvane Cavalcante de Oliveira. *Issues in Large Scale Collaborative Virtual Environments*. PhD thesis, Ottawa-Carleton Institute of Electrical and Computer Engineering, 2001.
- [32] N.D. de Oliveira, J.C. Georganas. Velvet: an adaptive hybrid architecture for very large virtual environments. *Communications, 2002. ICC 2002. IEEE International Conference on*, 4:2491–2495, 2002.
- [33] Dawei Ding and Miaoling Zhu. A model of dynamic interest management: interaction analysis in collaborative virtual environment. In *VRST '03: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 223–230, New York, NY, USA, 2003. ACM Press.

- [34] Ta Nguyen Binh Duong and Suiping Zhou. A dynamic load sharing algorithm for massively multiplayer online games. In *Networks, 2003. ICON2003. The 11th IEEE International Conference on*, pages 131–136, 2003.
- [35] Hugh Fisher. Multicast issues for collaborative virtual environments. *IEEE Computer Graphics and Applications*, 22(5):68–75, 2002.
- [36] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.
- [37] Martin Fowler. Domain specific languages. <http://www.martinfowler.com/bliki/DomainSpecificLanguage.html>, August 2007.
- [38] Patrik Fuhrer, Ghita Kouadri Mostéfaoui, and Jacques Pasquier-Rocha. Madviworld: a software framework for massively distributed virtual worlds. *Software Practice and Experience*, 32(7):645–668, May 2002.
- [39] T.A. Funkhouser. RING: a client-server system for multi-user virtual environments. *Proceedings of the 1995 symposium on Interactive 3D graphics*, 1995.
- [40] Athanasios Gaitatzes, Dimitrios Christopoulos, and Maria Roussou. Reviving the past: cultural heritage meets virtual reality. In *VAST '01: Proceedings of the 2001 conference on Virtual reality, archeology, and cultural heritage*, pages 103–110, New York, NY, USA, 2001. ACM Press.
- [41] J. J. Gibson. The theory of affordances. In R. Shaw & J. Bransford, editor, *Perceiving, acting and knowing*. Erlbaum, 1977.
- [42] Paul Graham. Programming bottom-up. <http://www.paulgraham.com/progbot.html>, August 2007.
- [43] C. Greenhalgh and S. Benford. Boundaries, Awareness and Interaction in Collaborative Virtual Environments. *Proceedings of the Sixth IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE Computer Society Cambridge, MA, pages 193–198, 1997.
- [44] Chris Greenhalgh. *Large Scale Collaborative Virtual Environments*. PhD thesis, University of Nottingham, 1997.
- [45] Chris Greenhalgh, Steve Benford, and Mike Craven. Patterns of network and user activity in an inhabited television event. In *VRST '99: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 34–41, New York, NY, USA, 1999. ACM Press.



- [46] Chris Greenhalgh and Steven Benford. *Massive: a collaborative virtual environment for teleconferencing*. *ACM Trans. Comput.-Hum. Interact.*, 2(3):239–261, 1995.
- [47] Irene Greif, editor. *Computer-supported cooperative work: a book of readings*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.
- [48] Seunghyun Han, Mingyu Lim, and Dongman Lee. Scalable interest management using interest group based filtering for large networked virtual environments. In *VRST '00: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 103–108, New York, NY, USA, 2000. ACM Press.
- [49] David Heinemeier Hansson. Active record. <http://wiki.rubyonrails.org/rails/pages/ActiveRecord>, August 2007.
- [50] David Heinemeier Hansson. Active support. <http://rubyforge.org/projects/activesupport/>, August 2007.
- [51] Larry F. Hodges, Benjamin Watson, G. Drew Kessler, Dan Opdyke, and Barbara O. Rothbaum. A virtual airplane for fear of flying therapy. *vrais*, 00:86, 1996.
- [52] Mojtaba Hosseini, Steve Pettifer, and Nicolas D. Georganas. Visibility-based interest management in collaborative virtual environments. In *CVE '02: Proceedings of the 4th international conference on Collaborative virtual environments*, pages 143–144, New York, NY, USA, 2002. ACM Press.
- [53] id Software. Doom. <http://www.idsoftware.com/games/doom/doom-ultimate/>, August 2007.
- [54] id Software. Quake. <http://www.idsoftware.com/games/quake/quake/>, August 2007.
- [55] id Software. Return to castle wolfenstein. <http://www.idsoftware.com/games/wolfenstein/rtcw/>, August 2007.
- [56] Takuji Iimura, Hiroaki Hazeyama, and Youki Kadobayashi. Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 116–120, New York, NY, USA, 2004. ACM Press.
- [57] Blizzard Entertainment Inc. World of warcraft. <http://www.worldofwarcraft.com/>, August 2007.
- [58] Linden Research Inc. Second life. <http://secondlife.com/>, August 2007.
- [59] Brian Ingerson, Clark Evans, and Oren Ben-Kiki. Yaml. <http://www.yaml.org/>, August 2007.
- [60] Geoffrey James. *The Tao of Programming*. Infobooks, 1986.

- [61] Edmund Weiner John Simpson, editor. *Oxford English Dictionary*, volume twenty volumes. Clarendon Press, second edition edition, 1989.
- [62] Levine J.R., Tony Mason, and Doug Brown. *Lex and Yacc*. O'Reilly, 1992.
- [63] Damien Katz. Couchdb. <http://couchdb.org/>, Augues 2007.
- [64] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall Professional Technical Reference, 1983.
- [65] Valerie D. Lehner and Thomas A. DeFanti. Distributed virtual reality: Supporting remote collaboration in vehicle design. *IEEE Computer Graphics and Applications*, 17(2):13–17, 1997.
- [66] Emmanuel Lety and Thierry Turetletti. Issues in designing a communication architecture for large-scale virtual environments. In *Networked Group Communication*, pages 54–71, 1999.
- [67] Emmanuel Léty, Thierry Turetletti, and François Baccelli. Score: a scalable communication protocol for large-scale virtual environments. In *Networking, IEEE/ACM Transactions on*, volume 12, pages 247–260, 2004.
- [68] Qingping Lin, Hoon Kang Neo, Liang Zhang, Guangbin Huang, and Robert Gay. Grid-based large-scale web3d collaborative virtual environment. In *Web3D '07: Proceedings of the twelfth international conference on 3D web technology*, pages 123–132, New York, NY, USA, 2007. ACM Press.
- [69] Elvis S. Liu, Milo K. Yip, and Gino Yu. Scalable interest management for multidimensional routing space. In *VRST '05: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 82–85, New York, NY, USA, 2005. ACM Press.
- [70] R. Bowen Loftin and Patrick J. Kenney. Training the hubble space telescope flight team. *IEEE Comput. Graph. Appl.*, 15(5):31–37, 1995.
- [71] Last.fm Ltd. Last.fm. <http://last.fm>, August 2007.
- [72] M. R. Macedonia, M. J. Zyda, D. R. Pratt, P. T. Barham, and S. Zeswitz. NPSNET: A network software architecture for large-scale virtual environment. *Presence*, 3(4):265–287, 1994.
- [73] Michael R. Macedonia, Donald P. Brutzman, Michael J. Zyda, David R. Pratt, Paul T. Barham, John Falby, and John Locke. Npsnet: a multi-player 3d virtual environment over the internet. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 93–ff., New York, NY, USA, 1995. ACM Press.
- [74] Michael R. Macedonia and Michael J. Zyda. A taxonomy for networked virtual environments. *Multimedia, IEEE*, 4:48–56, 1997.


- [75] Tony Manninen. Rich interaction in networked virtual environments. In *ACM Multimedia*, pages 517–518, 2000.
- [76] Michal Masa and Jiří Žára. Generalized interest management in virtual environments. *Collaborative virtual environments*, 2002.
- [77] Thomas W. Mastaglio and Robert Callahan. A large-scale complex virtual environment for team training. *Computer*, 28(7):49–56, 1995.
- [78] Maja Matijasevic. A review of networked multi-user virtual environments. Technical Report TR97-8-1, Center for Advanced Computer Studies, Virtual Reality and Multimedia Laboratory, University of Southwestern Louisiana, USA., 1997.
- [79] Yukihiro Matsumoto. Treating code as an essay. In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, chapter 29, pages 477–481. O'Reilly, 2007.
- [80] J. Maxfield, T. Fernando, and P. Dew. A distributed virtual environment for concurrent engineering. *IEEE Annual Virtual Reality International Symposium, Triangle Park, NC, USA*, 00:162, 1995.
- [81] W. Dean McCarty, Steven Sheasby, Philip Amburn, Martin R. Stytz, and Chip Switzer. A virtual cockpit for a distributed interactive simulation. *IEEE Comput. Graph. Appl.*, 14(1):49–54, 1994.
- [82] Erin McKean, editor. *New Oxford American Dictionary*. Oxford University Press, second edition edition, May 2005.
- [83] M. Meehan. Survey of multi-user distributed virtual environments. *course notes: "Developing Shared Virtual Environments"*. *SIGGRAPH*, 99, 1999.
- [84] Rob Minson and Georgios Theodoropoulos. An adaptive interest management scheme for distributed virtual environments. In *PADS'05: Principles of Advanced and Distributed Simulation, 2005*, pages 273–281, June 2005.
- [85] Katherine L. Morse. Interest management in large-scale distributed simulations. Technical Report ICS-TR-96-27, Department of Information & Computer Science, University of California, Irvine, 1996.
- [86] S. R. Musse and D. Thalmann. A model of human crowd behavior: Group inter-relationship and ... In *Computer Animation and Simulations '97, Proc Computer Animation and Simulations '97, Proc*, pages 39–51, 2002.

- [87] Soraia R. Musse, Christian Babski, Tolga Çapın, and Daniel Thalmann. Crowd modelling in collaborative virtual environments. In *VRST '98: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 115–123, New York, NY, USA, 1998. ACM Press.
- [88] Bonnie Nardi and Justin Harris. Strangers and friends: collaborative play in world of warcraft. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 149–158, New York, NY, USA, 2006. ACM Press.
- [89] Beatrice Ng, Rynson W. H. Lau, Antonio Si, and Frederick W. B. Li. Multiserver support for large-scale distributed virtual environments. *IEEE Transactions on Multimedia*, 7(6):1054–1065, 2005.
- [90] Federation of American Scientists. Spacecast 2020 glossary of terms. <http://www.fas.org/spp/military/docops/usaf/2020/app-v.htm>, August 2007.
- [91] The Ruby on Rails Community. Ruby on rails. <http://rubyonrails.com/>, August 2007.
- [92] Cory R. Ondrejka. Aviators, Moguls, Fashionistas and Barons: Economics and Ownership in Second Life. *SSRN eLibrary*, 2004.
- [93] Oracle. Oracle spatial. <http://www.orafaq.com/faq/spatial>.
- [94] Sungju Park, Dongman Lee, Mingyu Lim, and Chansu Yu. Scalable data management using user-based caching and prefetching in distributed virtual environments. In *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 121–126, New York, NY, USA, 2001. ACM Press.
- [95] Carolina Cruz-Neira Patrik Hartling, Chris Just. Distributed virtual reality using octopus. In *Virtual Reality, 2001. Proceedings. IEEE*, pages 53–60, 2001.
- [96] Jon Pearce. *Programming and Meta-programming in Scheme*. Springer-Verlag New York Inc., 1997.
- [97] Nicholas J. Pioch, Bruce Roberts, and David Zeltzer. A virtual environment for learning to pilot remotely operated vehicles. In *VSMM '97: Proceedings of the 1997 International Conference on Virtual Systems and MultiMedia*, page 218, Washington, DC, USA, 1997. IEEE Computer Society.
- [98] David R. Pratt. *A Software Architecture for the Construction and Management of Real Time Virtual Worlds*. PhD thesis, Naval Postgraduate School, 1993.
- [99] J. Purbrick and C. Greenhalgh. Extending locales: Awareness management in massive-3. In *VR2000*, pages 287–287, 2000.

- [100] E. Reid. Cultural formations in text-based virtual realities. Technical report, Department of History, University of Melbourne, 1994.
- [101] Abdennour El Rhalibi, Madjid Merabti, and Yuanyuan Shen. Aoim in peer-to-peer multiplayer online games. In *ACE '06: Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology*, page 71, New York, NY, USA, 2006. ACM Press.
- [102] P. Rosendale and Cory. Ondrejka. Enabling player-created online worlds with grid computing and streaming. Technical report, Gamasutra, 2003.
- [103] Mark A. Sagar, David Bullivant, Gordon D. Mallinson, and Peter J. Hunter. A virtual environment and model of the eye for surgical simulation. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 205–212, New York, NY, USA, 1994. ACM Press.
- [104] Masatoshi SEKI. Erb — ruby templating. <http://ruby-doc.org/stdlib/libdoc/erb/rdoc/>, August 2007.
- [105] Sandeep Singhal and Michael Zyda. *Networked virtual environments: design and implementation*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.
- [106] S.K. Singhal and D.R. Cheriton. Using projection aggregations to support scalability in distributed simulation. In *ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, page 196, Washington, DC, USA, 1996. IEEE Computer Society.
- [107] Rory John Stuart. *The Design of Virtual Environments*. Barricade Books Inc., 2001.
- [108] Gary Tan and Xu Liang. An Agent-based Data Filtering Mechanism for High Level Architecture. *SIMULATION*, 76(6):329–344, 2001.
- [109] Gary Tan, YuSong Zhang, and Rassul Ayani. A hybrid approach to data distribution management. In *DS-RT '00: Proceedings of the Fourth IEEE International Workshop on Distributed Simulation and Real-Time Applications*, page 55, Washington, DC, USA, 2000. IEEE Computer Society.
- [110] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 61–70, New York, NY, USA, 1991. ACM Press.
- [111] Why the Lucky Stiff. Why's poignant guide to ruby. <http://poignantguide.net/ruby/>, August 2007.
- [112] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby, The Pragmatic Programmers' Guide*. The Pragmatic Programmers, second edition edition, 2005.

- [113] Dave Thomas and David Heinemeier Hansson. *Agile Web Development with Rails*. The Pragmatic Programmers, second edition, 2006.
- [114] H. Tramberend. Avocado – a distributed virtual environment framework. In *IEEE Virtual Reality*, pages 14–21, 1999.
- [115] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.
- [116] Valve. Counter strike. [www.counter-strike.net/](http://www.counter-strike.net/), August 2007.
- [117] W3C. Vtml virtual reality modeling language. <http://www.w3.org/MarkUp/VRML/>, August 2007.
- [118] G. Gary Wang. Definition and review of virtual prototyping. *Journal of Computing and Information Science in Engineering*, 2(3):232–236, 2002.
- [119] R.C. Waters, D.B. Anderson, J.W. Barrus, D.C. Brogan, M.A. Casey, S.G. McKeown, T. Nitta, I.B. Sterns, and W.S. Yerazunis. Diamond Park and Spline: A Social Virtual Reality System with 3D Animation, Spoken Interaction, and Runtime Modifiability. *Presence: Teleoperators and Virtual Environments*, 6(4):461–480, 1997.
- [120] Wikipedia. Definition of virtual reality (redirected from virtual environment). [http://en.wikipedia.org/wiki/Virtual\\_environment](http://en.wikipedia.org/wiki/Virtual_environment), August 2007.
- [121] Bruce Sterling Woodcock. An analysis of mmog subscription growth. <http://www.mmogchart.com>, August 2007.
- [122] Yahoo! del.icio.us. <http://del.icio.us/>, August 2007.
- [123] Yahoo! flickr. <http://flickr.com>, August 2007.

# Colophon

This document was produced on an Mac running OS X. X<sub>Ǝ</sub>T<sub>E</sub>X was used for typesetting, TextMate for editing, BibDesk for managing the BibT<sub>E</sub>X references, Skim for previewing, Omnigraffle for creating diagrams, and Subversion for versioning. X<sub>Ǝ</sub>T<sub>E</sub>X is based on the T<sub>E</sub>X typesetting system by Donald Knuth. The standard font used throughout is Computer Modern Roman, and the monospace font used for code samples and algorithms is DejaVu Sans Mono.